B.Sc. Michael Spranger

Konzeption, Entwurf und Evaluierung eines dynamischen Konfigurations- und Parametrisierungs-Frameworks zur Unterstützung des Integrationsprozesses in heterogenen IT-Landschaften

Master Thesis

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Fakultät Mathematik, Naturwissenschaften, Informatik

Chemnitz, Januar 2012

B.Sc. Michael Spranger

Konzeption, Entwurf und Evaluierung eines dynamischen Konfigurations- und Parametrisierungs-Frameworks zur Unterstützung des Integrationsprozesses in heterogenen IT-Landschaften

eingereicht als

Master Thesis

an der

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Mathematik, Naturwissenschaften, Informatik

Chemnitz, Januar 2012

Erstprüfer:	Prof. Dr. Ing. Wilfried Schubert
Zweitprüfer:	Dipl. Inf. Chris Hübsch

vorgelegte Arbeit wurde verteidigt am:

Bibliografische Beschreibung

Spranger, Michael:
Konzeption, Entwurf und Evaluierung eines dynamischen Konfigurations- und Parametrisierungs-Frameworks zur Unterstützung des Integrationsprozesses in heterogenen IT-Landschaften. - 2011. - 103 S. Mittweida, Hochschule Mittweida, Fachbereich Mathematik, Naturwissenschaften, Informatik, Master-Thesis, 2011

Referat

Conception, design and evaluation of a dynamically Framework for configuration and parameterization for the purpose of technical assistance during the integration process within heterogeneous IT landscapes.
Ziel der Masterarbeit ist die Erstellung eines offenen Konzeptes zur Konfiguration und Parametrisierung von Schnittstellen beliebiger Software-Anwendungen, insbesondere integrierter Portal-Lösungen, mit dem Ziel der Datenintegration unter Nutzung und evolutionärer Weiterentwicklung des Remarc® Enterprise Integration Framework (EIF). Hierfür werden verschiedene Programmier-Paradigmen aus dem Enterprise-Umfeld und deren Umsetzung innerhalb der Eclipse-Plattform betrachtet und die erarbeiteten Technologien zur Entwicklung einer Architektur für ein Konfigurations-Framework herangezogen, das als Teil des EIF den bidirektionalen Datenaustausch zwischen verschiedenen Anwendungen optimal unterstützt.
Die Evaluation des erarbeiteten Konzeptes soll mit Hilfe eines Prototyps am Beispiel der Integration des CAD-Systems Siemens NX, des PDM/PLM-Systems Siemens Teamcenter und des CSM-Systems Remarc® erfolgen.

Inhaltsverzeichnis

[1] vergleiche Dokumentation zu Siemens Teamcenter 8.3, Teil: „Getting Started with Customization“

Listings

Abbildungsverzeichnis

Abkürzungsverzeichnis

API ..... **A**pplication **P**rogramming **I**nterface
CAD ..... **C**omputer **A**ided **D**esign
CAE ..... **C**omputer **A**ided **E**ngineering
CAM ..... **C**omputer **A**ided **M**anufacturing
cPDm ..... **c**ollaborative **P**roduct **D**ata **m**anagement
CSM ..... **C**omponent **S**upply **M**anagement
DSL ..... **D**omain **S**pecific **L**anguage
EIF ..... **E**nterprise **I**ntegration **F**ramework
EMF ..... **E**clipse **M**odeling **F**ramework
GMF ..... **G**raphical **M**odeling **F**ramework
HTTP ..... **H**yper **T**ext **T**ransfer **P**rotocol
HTTP ..... **H**yper**T**ext **T**ransfer **P**rotocol
JVM ..... **J**ava **V**irtual **M**achine
LOV ..... **L**ist **O**f **V**alues
MCAD ..... **M**echanical **C**omputer **A**ided **D**esign
MDSD ..... **M**odel **D**riven **S**oftware **D**evelopment
MKS ..... **M**ehr**k**örper**s**imulation
OSGi ..... OSGi-Alliance früher(**O**pen **S**ervice **G**ateway **i**nitiative)
PDM ..... **P**roduct **D**ata **M**anagement
PLM ..... **P**roduct **L**ifecycle **M**anagement
ROI ..... **R**eturn **O**n **I**nvestment
SLA ..... **S**ervice **L**evel **A**greement
TCP ..... **T**ransmission **C**ontrol **P**rotocol
XML ..... e**X**tensible **M**arkup **L**anguage

1 Einleitung

1.1 Auf dem Weg zum Enterprise Integration Framework

Die Integration verschiedenster Softwareprodukte unter einer gemeinsamen Plattform ist von entscheidender strategischer Bedeutung in fast jedem wirtschaftlichen Sektor. Durch die Vermeidung von Prozess- und Datenbrüchen kann die Konsistenz der Produktdaten über den gesamten Produktlebenszyklus sichergestellt werden. Ein Framework zur Steuerung von Softwarewerkzeugen, welches derartige Prozess- und Datenbrüche überbrückt und damit die Grundlage zur Integration heterogener IT-Landschaften, wie sie beispielsweise im Maschinenbau anzutreffen sind, bildet, wurde in der Bachelorarbeit: „Entwurf und prototypische Implementierung eines Domain-Frameworks zur Integration und Steuerung von Softwarekomponenten am Beispiel des Modelldatenaustauschs zwischen diversen CAD-Systemen“ im Dezember 2009 entwickelt [Spr09]. In der Folge wurde das System weiter verallgemeinert. Die zentrale Bereitstellung der Datenmodelle des Zielsystems Remarc® CSM und des entwickelten Frameworks als *Distributed Shared Model*, die das unternehmensweite Arbeiten mit einem gemeinsamen Modell ermöglicht, war der entscheidende Schritt in die Enterprise-Welt. Im Rahmen eines Forschungsprojektes wird derzeit die Anwendbarkeit des entstandenen *Enterprise Integration Frameworks (EIF)* auf die IT-Welt des Facility-Managements, evaluiert.

1.2 Ziel der Arbeit

Die Integration diverser Werkzeuge, insbesondere des PDM/PLM-Portals Siemens Teamcenter, zeigte die Schwächen des bisherigen Integration Frameworks. Soll bei-

spielsweise ein Normteil von Remarc® CSM nach Teamcenter übertragen werden, so muss ein bidirektionales Mapping zwischen den verschiedenen Datenmodellen erfolgen, denn es soll einerseits ein neues Objekt im PDM-System angelegt oder eine neue Revision eines vorhandenen Objektes erzeugt werden, andererseits die Rückmeldung über vergebene IDs für spätere Zuordnungen an das CSM-System zurückgegeben werden. Hierfür ist vor allem die Kenntnis beider Datenmodelle erforderlich. Jedes dieser Modelle kann von hoher Komplexität sein, so dass das Konfigurieren des Integrationsworkflows eine ziemlich fehleranfällige und komplizierte Aufgabe darstellen kann. Es soll deshalb Ziel der vorliegenden Arbeit sein, ein hochdynamisches Framework zu entwickeln, welches als eigenständige Schicht im EIF diese anspruchsvolle Aufgabe erfüllt. Dabei soll es den Benutzer vor allem dabei unterstützen, korrekte Werte für ein erfolgreiches bidirektionales Mapping von Datenmodellen verschiedener zu integrierender Applikationen anzubieten bzw. dynamisch zu ermitteln. Dies stellt jedoch nur einen besonderen Anwendungsfall dar. Das Framework soll darüber hinaus allgemein in der Lage sein, Daten aus unterschiedlichsten Quellen zu beziehen und diese zu integrierenden Werkzeugen zur Parametrisierung und Konfiguration ihrer Schnittstellen zur Verfügung zu stellen. Einen weiteren Schwerpunkt bildet die Möglichkeit, Abhängigkeiten zwischen verschiedenen Konfigurationswerten zu definieren und diese zur Laufzeit aufzulösen.
Die Tragfähigkeit des erarbeiteten Konzeptes soll am Beispiel des in Abbildung 1.1 dargestellten Integrationsworkflows zwischen Remarc® CSM, Siemens NX und Siemens Teamcenter prototypisch nachgewiesen werden.
Das zu entwickelnde Framework soll explizit keine eigenen Werkzeuge zur Transformation von Daten mit dem Ziel der Datenintegration konzipieren oder gar implementieren. Davon unberührt bleibt die Erfordernis der prototypischen Implementierung der neuen Schnittstellen zum Zweck der Anbindung der für den Funktionstest notwendigen Applikationen. Im Einzelnen soll ein in Remarc® CSM angelegtes

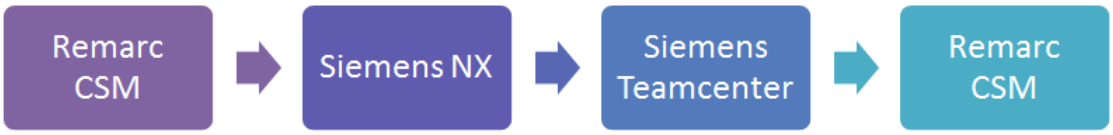

Abbildung 1.1: Ziel-Workflow

DIN-Normteil mit dem Zusatzattribut Werkstoff angereichert, in native 3D-Modelle

des CAD-Systems NX transformiert und anschließend ins PDM-System Teamcenter importiert werden. Es wird für den Prototyp mindestens das Default-Datenmodell von Teamcenter vorausgesetzt, mit der Erweiterung für ein Objekt einen Werkstoff aus einer Werteliste(LOV) zuweisen zu können. Der Werkstoff soll aus einer LOV des PDM-Systems entnommen werden. Nach erfolgreichem Import soll die angelegte Item-ID dem CSM-System zurückgemeldet und dem dortigen Datensatz hinzugefügt werden.

1.3 Aufbau der Arbeit

In einem ersten Schritt sollen die für die Evaluierung benötigten Zielsysteme vorgestellt werden, wobei besonderen Wert auf die durch sie repräsentierten Schlüsseltechnologien gelegt werden wird. Das dort erarbeitete Wissen ist für das Verständnis des betrachteten Integrationsprozesses essentiell. Diesen Betrachtungen folgt eine Diskussion von Programmierparadigmen für Unternehmensanwendungen und deren Umsetzung innerhalb der Eclipse-Plattform. Diese Paradigmen sollen bei der Entwicklung des Konfigurations-Frameworks zum Einsatz kommen. Das darauf folgende Kapitel setzt sich kritisch mit der Architektur der in der bereits angeführten Bachelorarbeit entwickelten Workflow-Engine auseinander und zeigt deren Schwachstellen auf. Anschließend werden maßgeblichen Neuerungen und die Notwendigkeit der Einführung eines Konfigurations-Framworks vorgestellt, die diese Schwachstellen beheben und die Transformation der Anwendung zum *Enterprise Integration Framework* ermöglichen. Das vorletzte Kapitel befasst sich mit der Konzeption und Entwicklung einer Architektur für das neue Framework. Die Arbeit schließt mit der Zusammenfassung des Ergebnisses und der visionären Betrachtung der weiteren Entwicklungsschritte, zum einen aus der Sicht des beauftragenden Unternehmens, zum anderen aus dem wissenschaftlichen Blickwinkel der Übertragbarkeit des Lösungskonzeptes auf weitere Problemfelder.

Die beiliegende CD enthält neben der Arbeit in digitaler Form, alle Quelltexte, der neuen Features organisiert in Eclipse-PDE[2]-Projekten.

[2] PDE - PlugIn-Development-Environment

2 Die Zielsysteme

Nur wer sein Ziel kennt,
findet den Weg.

(Lao Zi, 6. Jhd.)

In diesem Kapitel werden die zum Zwecke der Evaluation des Konzeptes zu integrierenden Softwaresysteme und die durch sie vertretenen Technologien kurz vorgestellt und Möglichkeiten zur Bedienung ihrer jeweiligen Schnittstellen diskutiert.

2.1 Component Supply Management (CSM)

Die Recherche von wiederverwendbaren Komponenten, insbesondere Standardbauteilen und -baugruppen, nimmt einen beachtlichen Teil ($\geq 20\%$) der Produktentwicklungszeit in Anspruch. Das betrifft nicht allein den Bereich der Konstruktion, vielmehr werden alle Beteiligten der Wertschöpfungskette durch diesen Umstand berührt. Zu allem Überfluss endet die erfolglose Suche dann nicht selten in der Neukonstruktion oder Anpassung ähnlicher Komponenten. Das verursacht Duplikate oder verschiedene Versionen von eigentlich standardisierten Teilen oder Baugruppen. Genau dieses Vorgehen vergrößert das Problem für alle folgenden Recherchen, so dass ein zunehmender Qualitätsverfall der Datenbasis erkennbar wird.

Definition 2.1.1 (CSM-System)**.** *„Eine CSM-Software ermöglicht die einfache Erzeugung, Umwandlung, Anreicherung, Klassifizierung, Verwaltung, Suche und Bereitstellung von wiederverwendbaren Inhalten.“*

Das beschleunigt auf der einen Seite den Entwicklungsprozess, indem die genannten Recherchezeiten gesenkt werden und spart dadurch auf der anderen Seite natürlich auch Entwicklungskosten.

2.1.1 Remarc® CSM

Das Komponenten-Framework Remarc® stellt eine Reihe von Werkzeugen zur Verfügung, die dazu geeignet sind Anwender im Konstruktionsprozess bei der kostengünstigen und effizienten Entwicklung neuer Produkte zu unterstützen. Neben der Bereitstellung von Norm- und Wiederholteilen stehen Funktionalitäten zum Erzeugen von Bauteildatensätzen, zum deduzierten Anlegen geometrischer Repräsentationen und zum Einbau in neue und existierende Baugruppen fast jedes beliebigen CAD-Systems zur Verfügung.
Eine Anzahl von Schnittstellen zu PLM- und ERP-Systemen ermöglicht die Daten-Integration in eine unternehmensweite Datenbasis. Die Grundlage hierfür bildet ein erweiterter Stammdatenbegriff, der neben den typischen ERP-Daten auch Geometrien und Klassifikationen umfasst.
Seine modulare Architektur verdankt das System vor allem der zugrunde liegenden *Eclipse Rich Client Platform*. Durch die konsequente Nutzung von Standards, wie OSGi und eine Vielzahl von Erweiterungen für verschiedenste Anwendungsfälle, unterstützt diese Plattform ideal die Entwicklung von hochkomplexen, modularen Systemen.
Aus Anwendersicht werden zwei Grundkonfigurationen unterschieden. Der *Remarc® Consumer* beinhaltet grundsätzliche Funktionen zum Auswählen und Einsetzen von Bauteilen in ein oder mehrere durch den Anwender spezifizierte CAD-Systeme. Der *Remarc® Creator* erweitert diese Basisfunktionalität um weitere administrative Funktionen, zu denen nicht zuletzt das Anlegen konkreter Teileklassen und zugehöriger Instanzen aus dem Bereich der Standard-, Katalog- und Werknormteile, Formelemente und Halbzeuge zählen. Abbildung 2.1 verdeutlicht diesen Rollencharakter, sowie den modularen Aufbau der Anwendung.

Der MappingEditor (Anhang A.1) als zentrale Komponente des Creators stellt umfangreiche Funktionen und Algorithmen zur Zuordnung von Merkmalen aus verschiedenen Quellen zu den Stammdatensätzen zur Verfügung, mit deren Hilfe in der Folge verschiedene Repräsentationen in Form von Metadaten, 3D-Produkt- und Leichtgewichts-Modellen, 2D-Zeichnungen oder sonstige Dokumente automatisch generiert werden können.

[3] Mit freundlicher Genehmigung ARC Solutions GmbH

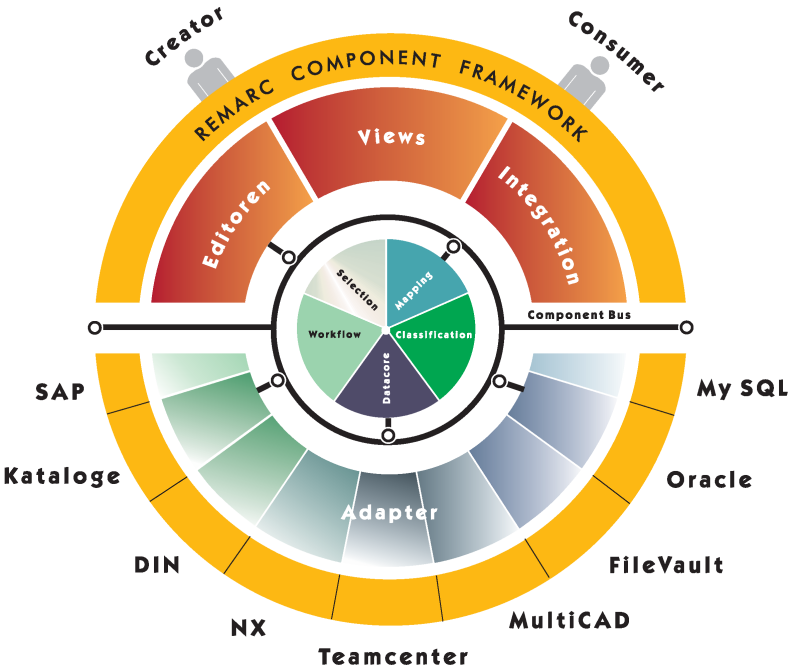

Abbildung 2.1: Remarc® Überblick[3]

2.1.2 Workflow-gesteuerte Integration

Das Anreichern von Daten mit Meta-Informationen, Datentransformationen und -transfers bilden die Kernfunktionalitäten des CSM-Systems Remarc®. Diese oft komplexen Abläufe wurden im Rahmen der o.g. Arbeit Workflow-basiert zu Prozesssketten verbunden und ereignisgesteuert zum Ablauf gebracht.
Mit Hilfe einer eigenen domänenspezifischen Sprache (DSL) können alle zur Datenintegration beitragenden Transfer- und Konvertierungsprozesse, nahezu beliebiger Komplexität, unter Nutzung eines grafischen Editors beschrieben werden. Abbildung A.2 im Anhang zeigt ein solches Modell (Workflow), welches dann rekursiv in neue Modelle integriert werden kann, um zum einen die Komplexität für den Nutzer zu senken, zum anderen aber auch an dieser Stelle dem Gedanken der Wiederverwendbarkeit Rechnung zu tragen. Ist ein Workflow hinreichend definiert und mit der aktuellen Produktkonfiguration lauffähig, wird er interpretiert und optimiert einem Scheduler zur Verfügung gestellt, der den Integrationsprozess anhand dieser Informationen steuert.

2.1.3 Konfiguration und Parametrisierung in Remarc® CSM

Um verschiedene Anwendungen auf Datenebene integrieren zu können, müssen einerseits von der Applikation zur Verfügung gestellte Schnittstellen genutzt (Adaptierung) und andererseits die unterschiedlichen Datenstrukturen aufeinander abgebildet werden (Mapping). Eine Regelmenge beschreibt, wie die korrekte Adaptierung bzw. das korrekte Mapping für jedes einzelne Datum aussieht. Abbildung 2.2 zeigt die bisher favorisierte Tabellen-basierte Lösung. Auf der linken Seite werden Schlüs-

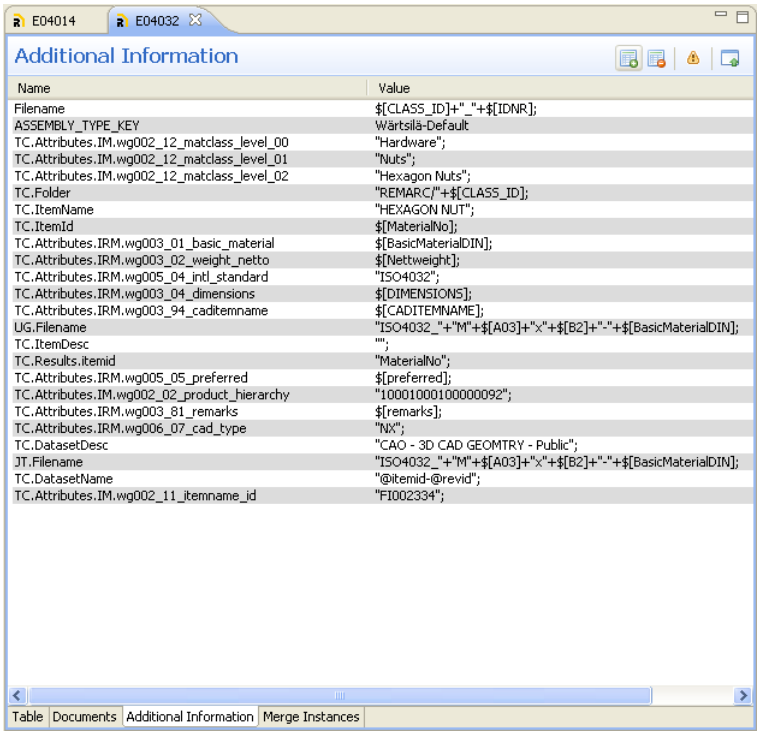

Abbildung 2.2: Remarc® Zusatzinformationen

sel definiert, welche das zugehörige Attribut im angeschlossenen System auf eindeutige Art und Weise beschreiben. Dabei wird versucht, die entsprechende Struktur mit Hilfe einer Punktschreibweise nach dem Muster

$$< System > . < Ebene1 > . < Ebene2 > ...$$

zu erschließen, wobei die Anzahl der Ebenen prinzipiell unbegrenzt ist.
In der zweiten Tabellenspalte werden die zugehörigen Werte in einer eigens hierfür

entwickelten externen DSL, mit welcher explizit Attribute des Datenmodells referenziert werden können, zugeordnet. Beispielsweise könnte so eine Dateinamenskonvention für eine Schraube aus dem Durchmesser und der Länge der jeweils aktuellen Teileklasseninstanz definiert werden.
Diese Assoziationen werden vom Scheduler genutzt, um die einzelnen Schnittstellen der Fremdsysteme korrekt zu konfigurieren bzw. parametrisieren. An dieser Stelle fällt bereits auf, dass die gesamte Konfiguration vor allem durch die kryptische Darstellung und die fehlende softwareseitige Unterstützung schwer zu lesen und äußerst fehleranfällig im Hinblick auf Schreibfehler und die Wahl der korrekten Schlüssel ist.

2.2 Product Data Management (PDM) und Product Lifecycle Management (PLM)

> „PDM ist das Management des Produkt- und Prozessmodells mit der Zielsetzung, eindeutige und reproduzierbare Produktkonfigurationen zu erzeugen.“ [ES09, S.34]

Ein *Produktmodell* bildet Produkte mit all ihren Informationen über den gesamten Lebenszyklus ab, so dass eine Art virtuelles Produkt entsteht, welches aus den nachfolgend aufgeführten beschreibenden Komponenten besteht:

- Produktstammsatz
- Produktstruktur
- technische Dokumente nach DIN 6789 und kommerzielle Dokumente
- Dokumentenstruktur

Außerdem dienen sogenannte Stücklisten dem Verwendungsnachweis und der Dokumentation der Produktkomposition.
Prozessmodelle beschreiben langfristige, kurzfristige oder temporäre technische und organisatorische Geschäftsabläufe. Sie werden oft auch als Workflow[4] bezeichnet.

[4] nicht zu verwechseln mit den Workflows aus dem Remarc® Enterprise Integration Framework

Kombiniert man beide Modelle erhält man ein *Konfigurationsmodell*, indem alle Informationen nach Status, Inhalt oder Version organisiert sind. [ES09]

> „Product Lifecycle Management (PLM) ist ein Konzept mit dem Ziel, den gesamten Produktlebenszyklus von der Idee bis zur Entsorgung eines Produktes durchgängig zu unterstützen, um produktrelevante Entscheidungen in den jeweiligen Phasen frühzeitig zu treffen. Hierzu zählt die gemeinsame Erarbeitung, Verwaltung, Kommunikation und Nutzung von produktbeschreibenden Informationen im gesamten Unternehmen.“ [VDM08, S.6]

Phasen in diesem Zusammenhang sind die nachfolgend aufgelisteten Abschnitte im Produktlebenszyklus, wobei jede einzelne durch bestimmte Aufgaben und Aktivitäten gekennzeichnet ist.

- Produktplanung/Design
- Produktentwicklung
- Vertrieb
- Auftragsspezifische Entwicklung
- Prozessplanung
- Beschaffung
- Produktion/Montage
- Service/Instandhaltung
- Umbau/Modernisierung
- Demontage/Entsorgung

Daneben gibt es Querschnittsprozesse, die alle Phasen berühren können. Hierzu zählen vor allem Änderungs-, Anforderungs-, Projekt-, Qualitäts- und Risikomanagement.

Das Product Data Management System (PDMS) bildet den Kern des PLM, d.h. alle Querschnittprozesse, Aufgaben und Aktivitäten der einzelnen Lebenszyklus-Phasen nutzen Funktionen des Systems, um auf dessen Datenbestand zugreifen zu

können. Aufgabe eines PDMS ist es daher Produktentwicklungssysteme und -daten zu integrieren. [VDM08]

2.2.1 Siemens Teamcenter

Einer aktuellen Marktstudie des anerkannten Managementberatungs- und Analystenhauses CIMdata zufolge wird die Nachfrage nach PLM-Software sowohl in Großunternehmen als auch im Mittelstand in den kommenden vier Jahren um weitere 50% ansteigen. Dabei teilen sich seit Jahren Siemens PLM, Dassault+IBM, SAP, PTC und Oracle die Marktspitze. Sie erwirtschaften gemeinsam etwa 55% des Marktvolumens. Unumstrittener Marktführer im Bereich cPDm mit der höchsten Marktpräsenz ist dabei Siemens PLM mit seinem PDM/PLM-System Teamcenter. [CIM11b] Abbildung 2.3 stellt die Marktsituation grafisch dar.

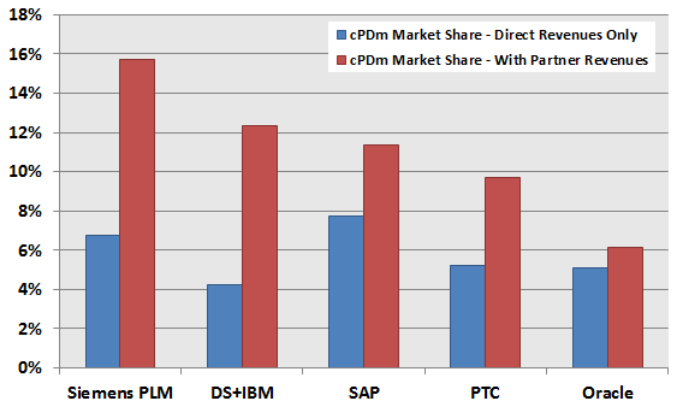

Abbildung 2.3: Marktpräsenz führender PDM-System-Anbieter [CIM11b, S.61]

2.2.2 Teamcenter Automation and Customization[5]

Die folgenden Ausführungen beziehen sich auf die aktuelle Version 8.3 von Teamcenter. Die Standard-Installation von Teamcenter ist bereits sehr mächtig und unterstützt das Management der Produktdaten über den gesamten Lebenszyklus. Neben

[5] vergleiche Dokumentation zu Siemens Teamcenter 8.3, Teil: „Getting Started with Customization“

den individuellen Anpassungen der Benutzerschnittstelle ist eine der herausragendsten Eigenschaften die Adaptionsfähigkeit für nahezu jeden Geschäftsprozess eines Unternehmens und die Fähigkeit Daten fremder Anwendungen zu integrieren. Im einzelnen können in den nachfolgend aufgeführten Bereichen Anpassungen vorgenommen werden:

- Das Verhalten von Teamcenter kann durch Integration externer Anwendungen, unter Benutzung des Integration Toolkit (ITK) oder Teamcenter Services, geändert werden.
- Das Aussehen der Anwendungsoberfläche kann, beispielsweise durch Änderung von Icons, angepasst werden.
- Durch Definition von Formularen kann die Darstellung des Datenbankinhaltes beeinflusst werden. Insbesondere kann festgelegt werden, **welche** Daten **wie** präsentiert werden sollen.
- Das Verhalten beim Datenaustausch kann regelbasiert angepasst werden.

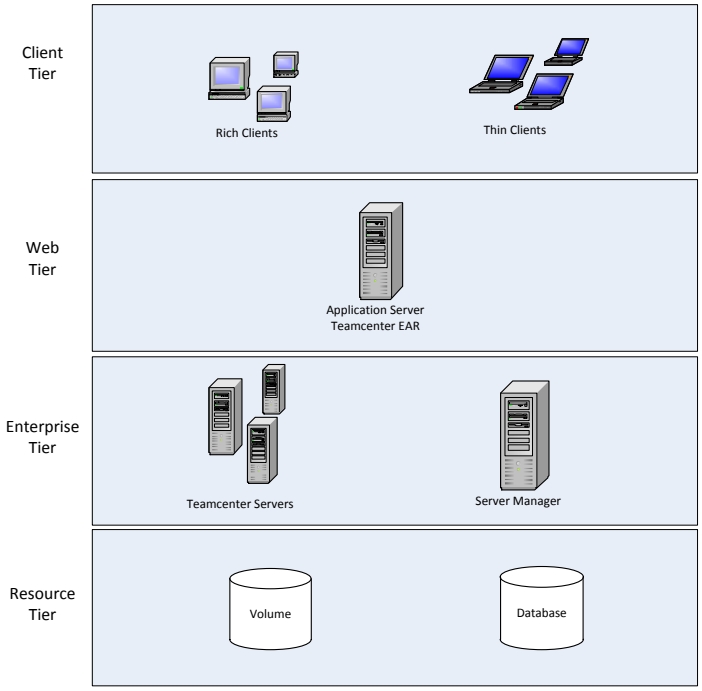

Abbildung 2.4: Teamcenter Schichten-Architektur

Teamcenter besitzt eine 4-Schicht-Architektur, wie in Abbildung 2.4 dargestellt.

Client tier: Diese Schicht beinhaltet die Teamcenter Clients. Es gibt Eclipse basierende *Rich Clients* und *Thin Clients*.

Web tier: Diese Schicht ist verantwortlich für die Kommunikation zwischen *Client tier* und *Enterprise tier*. Es handelt sich um eine Java Anwendung in einem *Java 2 Enterprise Edition* (J2EE) Application Server.

Enterprise tier: Diese Schicht holt und speichert Daten in der Datenbank. Der Server-Pool-Manager verwaltet Teamcenter-Server-Prozesse in einem konfigurierbaren Prozesspool. Hier können serverseitige Anpassungen mittels ITK-Programmierung vorgenommen werden.

Resource tier: Diese Schicht beinhaltet den Datenbank-Server, Volumes und File-Server mit einem entsprechenden Management-System (FMS).

Grundsätzlich besteht aus historischen Gründen zusätzlich noch die Möglichkeit das Portal als 2-Tier-Architektur zu betreiben, was aber in den aktuellen Versionen nicht mehr empfohlen wird.
Teamcenter unterscheidet grundsätzlich vier API's, um externe Anwendungen zu integrieren, bzw seinen Funktionsumfang durch externe Programmierung zu erweitern oder anzupassen.

Teamcenter Services: Teamcenter stellt eine serviceorientierte Architektur (SOA) zur Verfügung, um die Kommunikation zwischen verschiedenen Anwendungen zu ermöglichen. Die Vorteile dieser Architektur werden im Kapitel 3.2 näher untersucht. Diese Programmierschnittstelle soll bevorzugt genutzt werden, befindet sich aber noch im Aufbau, d.h. es kann möglicherweise noch nicht jede Interaktion programmatisch umgesetzt werden.

Integration Toolkit (ITK): Diese API ist zur serverseitigen Programmierung bestimmt und ermöglicht ausschließlich Low-Level-Zugriffe, von denen wesentlich mehr Aufrufe zur Realisierung einer bestimmten Funktion notwendig sind als bei den High-Level-API's, Teamcenter Services, Application Interface Web Service, Application Integration Environment. ITK enthält im Moment als einzige API den vollen Funktionsumfang, der auch interaktiv adressierbar ist. Existiert in den High-Level-Interfaces eine benötigte Funktion noch nicht, dann ist die

Entwicklung eines entsprechenden Services mit Hilfe des ITK das empfohlene Vorgehen.

Application Interface Web Service (AIWS): Diese API ermöglicht Anwendungen Daten auszutauschen, welche sich im selben Kontext befinden. Hierfür muss ein entsprechender AIWS-Proxy-Client in der externen Anwendung installiert werden. Diese API ist als nicht mehr zu verwenden markiert und existiert nur noch aus Kompatibilitätsgründen.

Generic Shell: Diese Schnittstelle ist ein Mechanismus, der externe Anwendungen ohne Programmierung einschließt und Kommandos zum Speichern von Daten abfängt und diese nach Teamcenter umleitet, so dass die Daten statt auf der Festplatte im PDM-System abgelegt werden. Generic Shell eignet sich jedoch nur für sehr einfache Integrationsaufgaben.

Da die Nutzung von Teamcenter Services empfohlen wird, soll zum Abschluss kurz qualitativ das Programmiermodell für elementare Operationen skizziert werden. Um CRUD-Operationen ausführen zu können, benötigt man eine Instanz der Klasse *Connection*, welche alle Verbindungs-beschreibenden Daten, wie Server-Name und Login-Daten, hält. Außerdem erzeugt man sich eine Instanz einer Klasse die das Interface *DataManagement* oder *FileManagement* implementiert. Diese Interfaces erlauben jeweils das programmatische ausführen von CRUD-Operationen für Metadaten bzw. für Dateien, die im Teamcenter verwaltet werden.

2.3 Computer Aided Design (CAD)

Hochkomplexe Software unterstützt weite Bereiche des zeitgemäßen Konstruktionsprozesses. Die Hauptanwendungsgebiete moderner CAD-Software liegen in den folgenden fünf Bereichen:

Problemdefinition umfasst die Festlegung von Funktion, Leistungsfähigkeit und äußerem Erscheinungsbild eines Produktes. Die Unterstützung erfolgt hier vor allem durch Bereitstellung von Teilebibliotheken und verwandten Entwürfen.

Geometrische Modellierung umfasst den Entwurf der Objektgeometrie. Die Softwareunterstützung ist hier vor allem in der Erzeugung einer Objektbeschreibung mit Hilfe von Primitiven und mathematischen Operationen zu finden. Diese Objektbeschreibung bildet die Basis für die Darstellung und Animation am Bildschirm.

Technische Analyse umfasst ein weites Spektrum an Analysemöglichkeiten, beispielsweise die Modellierung mit Hilfe finiter Elemente oder thermische und kinetische Berechnungsverfahren. Im Grunde gehören hierzu auch Simulationen, die jedoch nicht zwangsläufig zum Funktionsumfang eines reinen CAD-Systems gehören. Moderne CAD-Systeme sind modular aufgebaut und können deshalb durchaus auch Module zur Simulation integrieren.

Konstruktionsbewertung umfasst die Sicherstellung der Einhaltung von objektspezifischen Kriterien. Sie extrahiert Erkenntnisse über Herstellbarkeit, Genauigkeit und Kinematik eines Objektes.

Automatischer Entwurf dient in erster Linie zur Erstellung detailgetreuer Fertigungszeichnungen. Dabei können ganze Zeichnungen, maßstabsgetreue Ansichten, Ausschnittsvergrößerungen und Schnittflächen automatisch aus bestehenden Modellen abgeleitet und bemaßt werden.

[RNS94]
Der hart umkämpfte Markt besonders an der Spitze der Hersteller von CAD-Software, hat in den letzten Jahren immer mehr dazu geführt, dass sich aus reinen CAD-Systemen ganze Portale entwickelt haben, welche auch vielen Aufgaben aus angrenzenden Bereichen des CAE, beispielsweise der NC-Programmierung und Simulation (CAM) oder Mehrkörpersimulation (MKS), gewachsen sind.

2.3.1 Siemens NX

NX ist eine integrierte Software-Suite der Siemens PLM Software, die CAD-, CAM- und CAE-Anwendungen umfasst. Es unterstützt das Management des Produktlebenszyklus vor allem in den Bereichen Produkt-Design, Dokumentation, Simulation und Fertigung (Manufacturing). Es ermöglicht insbesondere die Erfassung und Wiederverwendung von Produkt- und Prozesswissen. Derartige Software-Suiten werden

auch als Multi-Discipline-CAD-Systeme (MCAD-MD) bezeichnet. Das anerkannte Managementberatungs- und Analystenhaus CIMData sieht in diesem Bereich Siemens NX neben CATIA von Dassault Systèmes und Creo Elements Pro von PTC an der Spitze des Weltmarktes.(Vergleiche Abbildung 2.5)

Ranking	Company	Brand	2010 Revenue ($M)
1	Dassault Systèmes	CATIA	159.6
2	Siemens PLM Software	NX	110.2
3	PTC	Creo Elements Pro	42.0

Abbildung 2.5: Ausschnitt aus dem Ranking der PLM-Anbieter [CIM11b, S.40]

2.3.2 NX Automation and Customization[6]

Siemens bietet mit *NX Open* eine Programmierplattform, basierend auf einem einheitlichen Objekt-Modell, zur Automation von NX für die Programmiersprachen C++, .NET und Java. Ausgehend von der *Open by Design* Philosophie werden alle Schritte bei der Konstruktion eines Produktes als Produktwissen im engeren Sinne aufgefasst und müssen deshalb, den PLM-Prinzipien folgend, vollständig konservierbar sein. Deshalb können die meisten Interaktionen, die der Konstrukteur innerhalb der Anwendung ausführt direkt als ausführbarer Quellcode gespeichert (aufgezeichnet) und sogar innerhalb der Anwendung als eine Art Makro ausgeführt (abgespielt) werden (Journaling). Um dies zu ermöglichen, muss die API prinzipiell nahezu 100% des Funktionsumfanges von NX über seine Programmierschnittstelle für den Fremdzugriff anbieten. Abbildung 2.6 zeigt einen repräsentativen Ausschnitt aus dem *Common Object Model*.

TaggedObject: Basisklasse für alle Entity-Klassen, wie *Line* oder *Extrude*. Sie repräsentiert persistente Entitäten im NX-Modell.

[6] vgl. Siemens NX Hilfe
[7] aus NX-Hilfe -> NX Open Programmer's Guide -> Common Object Model

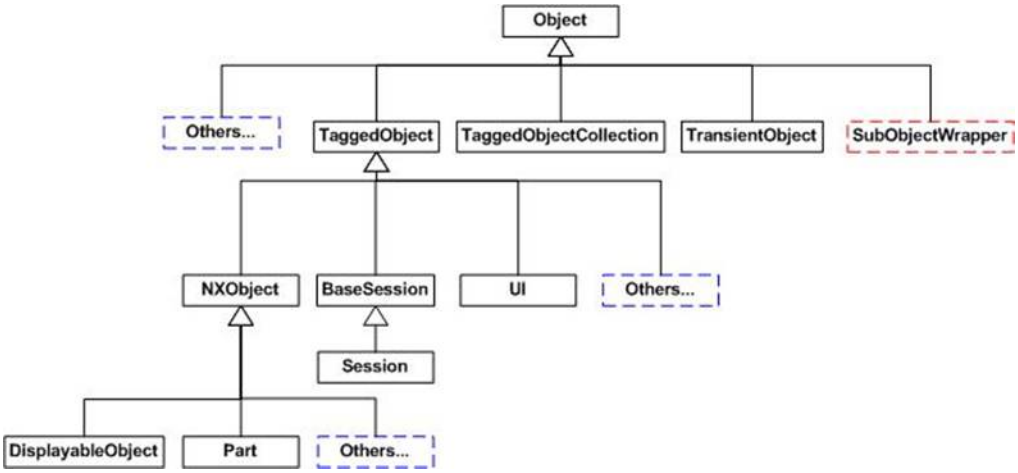

Abbildung 2.6: NX Open Common Object Model[7]

Session und UI: Sie dienen als Gateway-Klassen für die API und referenzieren direkt oder indirekt per Methode oder Eigenschaft alle anderen Objekte. Auf UI-Funktionalität kann selbstverständlich nur innerhalb des NX-UI zugegriffen werden.

Part: Repräsentiert ein *NX-Part*, also ein Einzelteil oder eine Baugruppe. Es enthält viele *TaggedObject-Collections*, die für den Zugriff auf die entsprechenden Objekte oder die Erzeugung neuer Objekte verantwortlich sind.

Dieser kleine Exkurs soll ausreichend sein, um einen Eindruck vom Umfang und den Möglichkeiten, welche die NX-API bietet, zu gewinnen und zu zeigen, dass NX Open dazu geeignet ist, die Anbindung von NX, unter Implementierung der neu entwickelten Schnittstellen, wie in Kapitel 1 beschrieben zu realisieren.

3 Eclipse Enterprise Architektur Paradigmen

Wer hohe Türme bauen will, muss lange beim Fundament verweilen.

(Anton Bruckner, 1824-1896)

Dieses Kapitel befasst sich mit Architektur Paradigmen zum Entwurf von Unternehmensanwendungen im Umfeld der Eclipse-RCP.

3.1 Model Driven Software Development (MDSD)

Definition 3.1.1. *„Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.“ [SVE07, S.11]*

Basis eines solchen Modells ist eine domänenspezifische Sprache (DSL). Als DSL bezeichnet man eine „kleine Programmiersprache“ mit Fokus auf einer bestimmten Domäne. Dadurch liegen sowohl Konzept als auch Notation so nah wie möglich am Denken in dieser Domäne. Man unterscheidet externe und interne DSLs. Während eine externe DSL einen völlig neuen Sprachraum definiert, ist eine interne DSL in einen existierenden Sprachraum eingebettet. Als Beispiel sei hier SQL genannt, eine externe DSL deren Fokus auf der Abfrage von Datenbanken liegt.
Das Gegenteil einer DSL ist die General Purpose Language (GPL). Mit GPLs, wie

der Programmiersprache Java, lässt sich grundsätzlich jedes Computer-Problem lösen. Die Frage nach der Effizienz der Lösung ist damit aber nicht geklärt.

Der potenzielle Nutzen modellgetriebener Softwareentwicklung im Allgemeinen und der dabei verwendeten DSL im Besonderen liegt vor allem in den nachfolgend aufgeführten Punkten:

- klarere Problembeschreibung mit daraus resultierendem besseren Verständnis der jeweiligen Domäne
- Konservierung von Geschäftswissen auf hohem semantischen Niveau
- Simplifizierung des Evolutionsprozesses
- Steigerung der Produktivität
- Erleichterung im Umgang mit großen Datenmengen
- Entwurf und Erhaltung einer einheitlichen Softwarearchitektur
- Verbesserung innerer Qualitätsmerkmale, insbesondere Korrektheit und Evolvierbarkeit

Dem gegenüber steht natürlich ein hoher initialer Aufwand bei der Entwicklung einer DSL, vor allem bei komplexen Domänen, sowie die Tatsache, dass meist nur ein Software-Rahmen generierbar ist. Außerdem ist die Validierung auf Modellebene nicht trivial und Fehler können oft erst zur Laufzeit aufgespürt werden.

3.1.1 Eclipse Modeling Framework (EMF)

Das EMF erweitert Eclipse um die Fähigkeit strukturierte, hochkomprimierte Datenmodelle zu entwickeln. Neben diversen Views und Editoren zur Modellierung abstrakter Syntaxen liefert das Framework auch Tools zur Quellcode-Generierung in Java, Runtime-Support, Transaktionsunterstützung, Persistenz, sowie zur Modell-Validierung, -Abfrage, -Suche und -Vergleich.[Ste+09]

3.1.2 Connected Data Objects (CDO)[8]

Voraussetzung für die Entwicklung von Enterprise-Lösungen ist die Ablage und Bereitstellung von Daten im unternehmensweiten Zugriff, so dass orthogonale Aspekte der Softwareentwicklung Berücksichtigung finden. Die EMF-Extension CDO stellt ein Distributed Shared Model Repository für EMF-Modelle zu Verfügung. Dabei bedient es, im Gegensatz zu einer reinen Datenbanklösung mit OR-Mapper, genau diese orthogonalen Aspekte. Insbesondere ist das Framework hochskalierbar, bietet Möglichkeiten zur verteilten Transaktionalität und Persistenz mit prinzipiell beliebigem Backend.

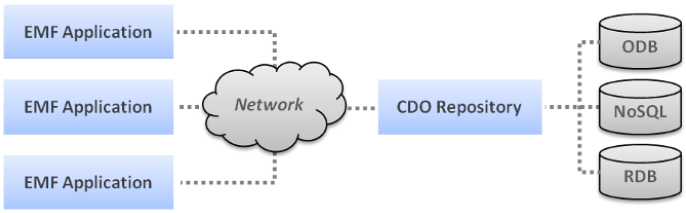

Abbildung 3.1: CDO Überblick [Ste]

Insbesondere die nachfolgend kurz diskutierten Hauptfunktionalitäten von CDO ebnen einer EMF-basierten Anwendung den Schritt in die Enterprise-Welt.

Persistence - CDO ermöglicht das Speichern eines Modells in prinzipiell alle Arten von Datenbanken. Anbieterspezifische Logik und Abhängigkeiten werden vollständig vor der Anwendung verborgen und ermöglichen so das leichte Austauschen des Backends.

Multi User Access - CDO erlaubt den gleichzeitigen Zugriff verschiedener Benutzer auf das Modell, indem es Sitzungen (*Sessions*)für jeden Client und jedes *Repository* erzeugt. Dabei werden verschiedene steckbare Transportprotokolle unterstützt. Für die nötige Sicherheit sorgt die Möglichkeit der Authentifizierung für jedes *Repository*.

Transactional Access - Der transaktionale Zugriff nach dem ACID-Prinzip wird wahlweise mit optimistischen oder pessimistischen Locking bis auf Objekt-Ebene realisiert. In einer verteilten Umgebung mit mehreren *Repositories* wird

[8] Vergleiche [Ste]

Transaktionalität durch ein *Zwei-Phasen-Commit-Protokoll* innerhalb einer sogenannten *XA Transaction* gewährleistet.

Transparent Temporality - Mit Hilfe sogenannter *Audit Views* können beliebige historische Versionen eines konsistenten *Object Model Graph* angezeigt werden.

Parallel Evolution - Verschiedene Versionen eines Objekt-Graphen können parallel in Branches entwickelt werden. Dieses Vorgehen ähnelt dem in den bekannten Versionsverwaltungssystemen für die Quellcode-Entwicklung.

Scalability - Die Fähigkeit mit prinzipiell beliebig großen Modell-Graphen umgehen zu können, wird durch Cachen einzelner Objekte erreicht, die auf Anforderung geladen wurden (*Lazy Loading*). Nicht mehr benötigte Objekte, entfernt der *Garbage Collector* der *JVM*, wenn der Speicher knapp wird. Verschiedene Vorlade-Strategien (*Prefetching*) optimieren den Zugriff auf den Objekt-Graphen weiter.

Thread Safety - CDO gewährleistet Thread-Sicherheit durch das Konzept multipler Transaktionen auf derselben Session. Dabei teilen sich alle dieselben Objektdaten bis eine Änderung durch einen der Threads erfolgt. Treten Konflikte auf können diese mit Hilfe individueller Konflikt-Handler gelöst werden.

Collaboration - Jede Anwendung wird über Remote-Änderungen am Objekt-Graphen benachrichtigt, d.h. werden von einem beliebigen Teilnehmer Änderungen zum CDO-Server committed, so wird das Objekt-Modell aller anderen Teilnehmer ebenfalls aktualisiert. Der Grad der Kollaboration kann durch steckbare Handler für das asynchrone CDO-Protokoll oder angepasste *Change Subscription Policies* eingestellt werden.

Data Integrity - Die Datenintegrität kann insbesondere durch referenzielle Integritäts- und Kreisreferenz-Prüfungen zur Commit-Zeit gesichert werden. Weitere Prüfmechanismen können durch Implementierung sogenannter *Write-Access-Handler* nachgerüstet werden.

Fault Tolerance - Fehlertoleranz kann einerseits durch Aufsetzen von *Fail-Over-Clustern* aus replizierenden *Repositories* unter Kontrolle eines *Fail-Over-Monitors*, andererseits durch Nutzung spezieller Session-Typen *(Fail-Over-Session, Reconnecting-Session)*, die dem Client das Halten seiner lokalen Kopie

des Objekt-Graphen über einen Bruch der Verbindung zum Server hinaus gestattet, realisiert werden.

Offline Work - Die Offline-Arbeit ist auf zwei verschiedenen Wegen möglich. Eine Möglichkeit besteht im *Check-Out* einer speziellen Version eines bestimmten Branches in einen lokalen CDO-Workspace. Der Umgang mit dem Objekt-Graphen ähnelt in diesem Fall der Arbeit mit einem Quellcode-Management-System wie SVN. Eine weitere Möglichkeit liegt in der Erzeugung eines sogenannten *Offline-Clone*. In diesem Fall wird der komplette Objekt-Graph inklusive aller Branches und Revisionen auf den Client repliziert und synchronisiert sich danach fortlaufend mit dem *Master-Repository*, sofern eine Netzwerkverbindung verfügbar ist.

3.2 Service Oriented Architecture

> „SOA führt ein Architekturmodell ein, das Effizienz, Agilität und Produktivität eines Unternehmens dadurch steigern will, dass die Lösungslogik im Wesentlichen durch Services dargestellt wird. Dies erleichtert die Umsetzung der strategischen Ziele, die sich mit dem serviceorientierten Computing verbinden.“[Erl08, S.52]

Dabei stellt SOA keine Technologie sondern lediglich ein Paradigma dar. Auch wenn derartige Architekturen in der überwiegenden Zahl der Fälle heute in Form von Webservices implementiert werden, ist dies nicht die einzige Art des serviceorientierten Computing. Die OSGi-Alliance[9] definiert eine Softwareplattform mit einem eigenen Konzept der Serviceorientierung, die einerseits den Entwurf offener, modularer und skalierbarer Anwendungen in besonderem Maße unterstützt, andererseits den Aufbau serviceorientierter Architekturen in unternehmensweiten Java-basierten IT-Landschaften ermöglicht.

[9] früher: *Open Services Gateway initiative*, `http://www.osgi.org`

3.2.1 Technische Konzepte

Im Folgenden sollen die wichtigsten technischen Aspekte serviceorientierter Architekturen betrachtet werden. Das technische Modell mit den wesentlichen Bausteinen einer SOA zeigt Abbildung 3.2.

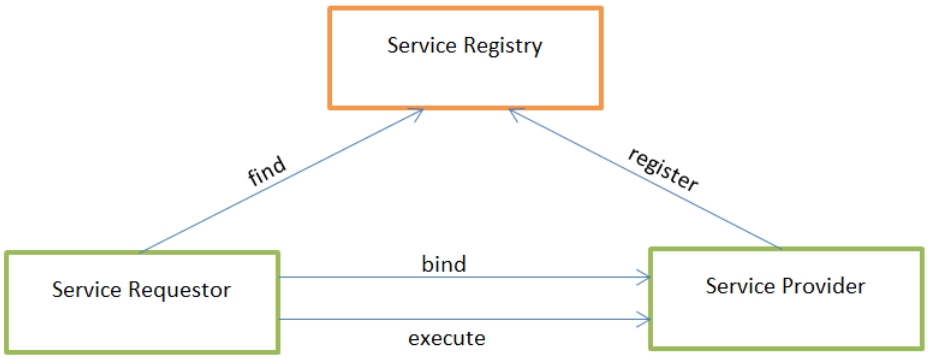

Abbildung 3.2: Wesentliche Komponenten einer SOA [ZG10, S.130]

Register/Publish Ein Dienstanbieter, der *Service Provider*, registriert/veröffentlicht seinen laut Vertrag bereitgestellten Dienst bei einer zentralen Registratur, der *Service Registry*. Vergleichbar ist dieser Vorgang mit dem Eintrag eines Dienstleistungsunternehmens in die Gelben Seiten.

Find Eine Softwarekomponente, der *Service Requestor*, die eine bestimmte Dienstleistung benötigt, stellt eine entsprechende Suchanfrage an die *Service Registry* und bekommt bestenfalls die Referenz eines passenden Dienstes zurück. Im betrachteten Beispiel sucht und findet ein Dienstleistungsnehmer also die Kontaktdaten eines passenden Unternehmens in den Gelben Seiten. Ist die Suche erfolgreich wird die Adresse des Unternehmens bekannt.

Bind Ist die Referenz eines passenden Dienstes bekannt, wird diese an den Funktionsaufruf gebunden. Dieses Vorgehen entspricht einem Vertrag zwischen dem Dienstleistungsnehmer und dem Unternehmen.

Execute Erfolgt der Funktionsaufruf, so wird der Dienst mit allen erforderlichen Parametern aufgerufen und liefert als Antwort das vertraglich vereinbarte Ergebnis. Der Dienstleitungsnehmer erteilt den Auftrag, liefert dabei alle zur

Bearbeitung notwendigen Informationen und erhält vom beauftragten Unternehmen das vereinbarte Produkt.

Services[10]

Als Services (Dienste) werden physikalisch unabhängige Softwareprogramme bezeichnet, die für einen zugewiesenen funktionalen Kontext, fachliche Funktionalität, basierend auf abstrahierten Geschäftsregeln, externen Consumer-Programmen zur Verfügung stellen. Jeder Service kann dabei prinzipiell eigene Entwurfsmerkmale besitzen. Seine Funktionalität wird meist durch einen veröffentlichten Service-Vertrag beschrieben. Dadurch bleiben technische Details und damit Plattformspezifika vor den Consumern verborgen. Es sollte klar sein, dass ein Service nicht nur eine Funktion zur Verfügung stellen kann, vielmehr fungiert er als Container für eine Reihe von Funktionalitäten in einem gemeinsamen Kontext.
Im Allgemeinen unterscheidet man anhand der Art der gekapselten Logik, dem Wiederverwendungspotenzial und der Bindung an eine spezifische Domäne die folgenden drei Service-Typen:

Entity-Services definieren ihren funktionalen Kontext und Abgrenzung basierend auf mindestens einer Business-Entity. Aufgrund der Abgeschlossenheit besitzt dieser Servicetyp ein hohes Wiederverwendungspotenzial.

Task-Services definieren ihren funktionalen Kontext und Abgrenzung direkt an einem bestimmten Geschäftsprozess (Business-Task). Diese enge Bindung vermindert die Möglichkeiten der Wiederverwendung, weshalb dieser Typ vor allem als Kompositions-Controller eingesetzt wird. Der Begriff *Kompositions-Controller* wird im Abschnitt *Service Composability* näher erläutert.

Utility-Services definieren ihren funktionalen Kontext und Abgrenzung unabhängig von der Geschäftslogik auf einer eigenen technologieorientierten Ebene. Das ermöglicht den höchsten Grad an Wiederverwendbarkeit.

[10] Vergleiche [Jos08; Erl08]

Standardized Service Contract[11]

Definition 3.2.1 (Servicevertrag)**.** *„Ein Vertrag für einen Service legt die Bedingungen seiner Pflichten fest und gibt sowohl technische Einschränkungen und Anforderungen als auch semantische Informationen an, die der Service-Owner veröffentlichen möchte.“ [Erl08, S.140]*

Serviceverträge bestehen neben der eigentlichen technischen Schnittstelle aus technischen Servicebeschreibungen, die zur Laufzeit verwendet werden und optionalen nichttechnischen Dokumenten, beispielsweise sogenannten *Service Level Agreements.*

Definition 3.2.2 (Entwurfsprinzip)**.** *„Im Zusammenhang mit der Erstellung von Lösungen ist ein Entwurfsprinzip eine empfohlene Richtlinie, um die Lösungslogik auf eine bestimmte Art und im Hinblick auf bestimmte Ziele zu gestalten.“ [Erl08, S.41]*

Serviceverträge werden durch die Nutzung von Entwurfsstandards, welche auf Entwurfsprinzipien beruhen, standardisiert. Dieses Vorgehen ist insofern erforderlich, da die Konsistenz der Services von entscheidender Bedeutung für die serviceorientierte Architektur ist. Einen Überblick über Arten der Standardisierung gibt die folgende Auflistung:

- Standardisierung des Funktionsaufrufes von Services
- Standardisierung der Datenrepräsentation von Services
- Standardisierung von Service-Policies

Service Loose Coupling[12]

Für eine serviceorientierte Architektur ist es von großer Bedeutung, dass der Abhängigkeitsgrad zwischen einzelnen Komponenten minimiert wird. [Jos08] definiert das Prinzip so:

[11] Vergleiche [Erl08, S.139 ff]
[12] Vergleiche [Erl08, S.175 ff]

Definition 3.2.3 (Lose Kopplung)**.** *„Unter loser Kopplung versteht man das Konzept, Abhängigkeiten zu minimieren. Je geringer die Abhängigkeiten, desto geringer die Auswirkungen von Veränderungen und Fehlverhalten. Je geringer aber die Auswirkung von Fehlern sind, desto fehlertoleranter sind wir, und je geringer die Auswirkungen von Veränderungen sind, desto flexibler sind wir.“[Jos08, S.22]*

Nach Art und Weise der Kopplung zwischen den Komponenten serviceorientierter Architekturen kann man verschiedene Typen identifizieren:

Logik-Vertrags-Kopplung Die Verwendung des *Contract-First-Ansatzes* gewährleistet die konsistente Umsetzung definierter Standards. Indem zuerst der Service-Vertrag formuliert wird, kann die Logik-Implementierung optimal auf die Belange des im Vertrag geregelten Services ausgerichtet werden.

Vertrag-Logik-Kopplung Die Umkehrung des vorangegangen Typs beschreibt ein häufig anzutreffendes Szenario beim Umstellen von Architekturen auf Serviceorientierung. Hier wird der Vertrag nachträglich von bereits vorhandenen Logik-Bausteinen, oftmals automatisiert, abgeleitet. Dieses Vorgehen führt zu einer engen Bindung des Vertrags an die Implementierung und in der Folge auch dazu, dass bei sich ändernder Implementierung auch Verträge angepasst werden müssen.

Vertrag-Technologie-Kopplung Bei der Nutzung proprietärer oder traditioneller Komponenten in der Service-Logik kann dies die Aufnahme technologiespezifischer Merkmale in den Service-Vertrag zur Folge haben, was den Dienstnutzer zwingt eben diese Technologie ebenfalls zu nutzen.

Vertrag-Implementierungs-Kopplung Das Einbinden von implementierungsspezifischen Details in den Servicevertrag führt zu dieser Art der Kopplung. Vor allem die Nutzung externer Ressourcen, wie physikalische Datenspeicher, erzeugen derartige Verbindungen. Der Nutzungsgrad derartiger Ressourcen gibt Auskunft über die Stärke der Kopplung.

Vertrag-Funktionalitäts-Kopplung Die Bindung der Funktionalität eines Services an einen bestimmten externen Funktionsbereich schränkt seine Allgemeingültigkeit in erheblichem Maße ein. Diese Art der Kopplung liegt also insbeson-

dere dann vor, wenn ein Service explizit erzeugt wurde, um einen bestimmten Geschäftsprozess zu unterstützen.

Lose Kopplung will alle Typen minimieren, mit Ausnahme der Logik-Vertrags-Kopplung. Die dort entstehende enge Kopplung ist durchaus sinnvoll, da sie der konsistenten Umsetzung der Serviceorientierung dient.
Die Voraussetzung für das Erfüllen einer derartigen Architektur-Forderung ist eine starke *Kohäsion* innerhalb einzelner zu verbindender Komponenten. Dabei bezeichnet der Begriff der *Kohäsion* in der Informatik den qualitativen Grad der Erfüllung einer zugeordneten Aufgabe oder Abbildung einer bestimmten Funktionseinheit. Übertragen auf Services lautet die Forderung also, dass jeder Service, die ihm zugeordnete Aufgabe, und nur diese, so gut und autark wie möglich erledigt.

Service Abstraction[13]

Der Begriff der Abstraktion bezeichnet im Zusammenhang mit Services ein Konzept, welches Entwurfs- bzw. Implementierungsdetails vor dem Nutzer eines Services (Consumer) verbergen soll, die für seine Nutzung nicht unbedingt erforderlich sind. In einen Service-Vertrag werden deshalb nur die unbedingt notwendigen Funktionen aufgenommen. Bei der Implementierung muss darauf geachtet werden, dass auch nur diese Funktionen Teil der öffentlichen API sind. Inhaltlich lassen sich die nachfolgend aufgeführten Service-relevanten Abstraktionstypen unterscheiden:

Abstraktion von technologischen Informationen meint das Verbergen von Informationen über Technologien, die zur Erstellung des Services verwendet wurden aber für die Dienstnutzung irrelevant sind, vor dem Nutzer. Hierzu zählen neben der verwendeten Programmiersprache beispielsweise auch Programmierschnittstellen von Drittanbietern.

Funktionale Abstraktion meint das Verbergen von Funktionen, die für den Consumer bei der Nutzung des Dienstes nicht von Interesse sind, weil sie beispielsweise nur einen Teil der Infrastruktur oder Hilfsfunktionen abbilden.

[13] Vergleiche [Erl08, S.219 ff]

Abstraktion der Programmlogik betrifft Entwurfsdetails der untersten Ebene, die im Allgemeinen vor den Consumern verborgen werden sollten. Beispielhaft seien hier Algorithmen, Fehlerbehandlung und Validierung genannt.

Abstraktion der Servicequalität betrifft Metainformationen über Regeln und Verhalten eines Services. Die meisten dieser Informationen können hinter der Abstraktionsebene verborgen werden. Es gibt jedoch qualitative Servicemerkmale die über den Service-Vertrag bekannt gemacht werden müssen, beispielsweise die Antwortzeiten des Services oder das Lastverhalten. Diese Informationen werden in der Regel in einem *Service Level Agreement (SLA)* veröffentlicht.

Es ist klar, dass diese Informationen jeweils unterschiedlichen Dokumenten zuzuordnen sind.

Service Reusability[14]

Wiederverwendbarkeit bezeichnet das Potential einer Software mehr als nur einmal eingesetzt werden zu können. Die tatsächliche mehrfache Nutzung ist hierfür nicht erforderlich. Im Zusammenhang mit serviceorientierten Computing soll die Servicelogik möglichst wiederholt verwendet werden, damit sich insbesondere Anfangsinvestitionen schneller amortisieren und die Agilität des Service Providers bezüglich neuer Anforderungen erhöht wird. Die Berücksichtigung dieses Prinzips ermöglicht direkt den Entwurf agnostischer Servicemodelle[15]. Services mit hohem Wiederverwendbarkeitsgrad sind vor allem gekennzeichnet durch eine hochgradig generische Servicelogik, die nebenläufige Zugriffe erlaubt, sowie generische und erweiterbare Verträge und einen agnostischen funktionalen Kontext. Es werden drei Grade der Wiederverwendbarkeit unterschieden:

Taktische Wiederverwendbarkeit beschränkt sich auf die Implementierung der momentan unbedingt notwendigen Funktionen mit der Möglichkeit der späteren Erweiterung.

[14] Vergleiche [Erl08, S.259 ff]

[15] hier vor allem als logikneutrale Servicemodelle mit weitgehend neutralem Funktionsrahmen zu verstehen

Zielgerichtete Wiederverwendbarkeit beschränkt die Implementierung auf die wichtigsten Funktionen, die im Rahmen einer vollständigen serviceorientierten Analyse ermittelt worden sind. Auch hier ist die nachträgliche Erweiterbarkeit Konzept.

Vollständige Wiederverwendbarkeit konzentriert sich nicht auf die Umsetzung von Anforderungen aus dem aktuellen Projekt, vielmehr wird versucht eine möglichst vollständige Palette von Funktionen, die im Rahmen der serviceorientierten Analyse ermittelt worden sind zu implementieren.

Welche dieser Grade zur Anwendung kommt ist entscheidend von den im Projektrahmen zur Verfügung stehenden Kapazitäten abhängig. Sind diese ausreichend für die Erstellung einer vollständigen serviceorientierten Analyse, sollte mindestens die zielgerichtete Wiederverwendbarkeit zum Einsatz kommen.

Service Autonomy[16]

Autonomie im Zusammenhang mit Services bezeichnet die Fähigkeit zumindest Funktionalitäten der Kernlogik unabhängig ausführen zu können. Dies wird insbesondere durch ein hohes Maß an Kontrolle (im besten Fall exklusive Kontrolle) über seine Laufzeitumgebung erreicht, was vor allem in komplexen Service-Inventaren erforderlich ist, um die Kontrolle über Abhängigkeiten und Ressourcen zu behalten. Im Umfeld einer OSGi-Laufzeitumgebung kommt diesem Prinzip eine besondere Bedeutung bei, wie im Abschnitt 3.2.3 diskutiert werden wird. Man kann zwei Formen der Autonomie unterscheiden:

Laufzeitautonomie bezeichnet die Kontrolle eines Service über seine Verarbeitungslogik zur Laufzeit. Ein hohes Maß an Laufzeitautonomie garantiert konsistent gute Performance bei gleichzeitig hoher Zuverlässigkeit durch berechenbares Verhalten sogar bei nebenläufigen Zugriffen.

Entwurfszeitautonomie bezeichnet die Fähigkeit eines Services sich im Laufe seines Lebenszyklus ändernden Anforderungen anzupassen. Vor allem Service-Consumer müssen eine starke Abhängigkeit zu den von ihnen verwendeten Diensten definieren um diese binden zu können. Diese Bindung schränkt die

[16] Vergleiche vgl.[Erl08, S.299 ff]

Fähigkeit des Services zur Weiterentwicklung ein, da er seine Vertragsverpflichtungen nicht verletzen darf. Ziel beim Entwurf muss es daher sein, den maximalen Grad an Entwurfszeitautonomie zu bewahren, um eine hohe Skalierbarkeit zu erreichen, auf geänderte Anforderungen reagieren und neue Technologien nutzen zu können.

Service Statelessness[17]

Die Skalierbarkeit eines Services hängt in besonderem Maße davon ab, wie er mit zustandsbehafteten Informationen umgeht. Es gilt, je geringer die Zustandsbehaftung des Services ist, desto besser ist seine Skalierbarkeit, da mit steigender Anzahl Klientenanfragen natürlich auch die Menge der zu verwaltenden und zu speichernden Zustandsdaten (Aktivitätsdaten) wächst und damit die gesamte Infrastruktur (Systemressourcen) belastet wird. Um Zustandslosigkeit eines Services zu erreichen, werden die Techniken Delegation und verzögerte Zustandsverwaltung eingesetzt. Das bedeutet, dass derartige Informationen regelmäßig temporär an einen anderen Ort der Architektur, beispielsweise eine Datenbank, verschoben werden (State Deferral), um sie erst bei Bedarf wieder zurückzuholen und zu interpretieren. Somit wird die Zustandsverwaltung an eine andere Komponente der Architektur delegiert.
Die Zustandslosigkeit von Services wirkt sich direkt auf die Prinzipien *Service Reusability* und *Service Autonomy* aus. Zum einen wird der Grad der Wiederverwendbarkeit erhöht, zum anderen sinkt durch die Verlagerung der Zustandsverwaltung die Gefahr von Abhängigkeiten zu anderen Teilen der Geschäftslogik. Im umgekehrten Fall kann natürlich die Autonomie, durch Abhängigkeiten zu Angeboten der Zustandsverwaltung über die Grenzen des Services hinaus, beeinträchtigt werden. Außerdem entstehen durch die Umsetzung dieses Prinzips und der damit verbundenen Steigerung der Verfügbarkeit natürlich auch Kosten in Form von niedrigerer Performance, verursacht durch das Verschieben und Interpretieren von Zustandsdaten.

[17] Vergleiche [Erl08, S.329 ff]

Service Discoverability[18]

Die wahrscheinlich wichtigsten Konzepte stecken naturgemäß im *Discovery*, also dem Prozess des Auffindens eines Services, und in der *Interpretation*, der Fähigkeit seinen Zweck und seine Fähigkeiten zu verstehen. Demzufolge sind die Entwurfsmerkmale Auffindbarkeit und Interpretierbarkeit von besonderem Interesse bei der Entwicklung von Services. Um diese Merkmale zu verbessern, werden sowohl *Funktionale Metainformationen* als auch *Metainformationen zur Servicequalität* benötigt. Das Auffinden selbst kann zur Entwurfszeit, also manuell, oder zur Laufzeit, durch dynamische Abfragen einer Service-Registry, wie von Universal Description, Discovery and Integration (UDDI) zur Verfügung gestellt, erfolgen.

Service Composability[19]

Beim Entwurf von Services ist darauf zu achten, dass sie effektiv in Kompositionen eingegliedert werden können. Dabei ist es unerheblich, ob eine solche Anforderung zu diesem Zeitpunkt besteht oder nicht. Insbesondere die Wiederverwendbarkeit profitiert von der sauberen Umsetzung dieses Prinzips.
In einer Komposition kann ein Service eine von zwei Rollen annehmen. Er kann einerseits in einer aktiven Rolle als *Kompositions-Controller* auftreten und Dienste bzw. Fähigkeiten anderer *Kompositionsmitglieder* anfordern, um seine eigene Logik ausführen zu können, andererseits eher passiv als *Kompositionsmitglied* seine Dienste und Fähigkeiten dem *Kompositions-Controller* zur Verfügung stellen (Abbildung 3.3). Selbstverständlich kann jedes Kompositionsmitglied als *Subcontroller* mit weiteren Services interagieren. Genau genommen handelt es sich bei der *Service-Komposition* um eine Komposition von Fähigkeiten. Während die Rollenzuweisung an einen Service von temporärer Natur ist, bleibt, aufgrund der Logik-Kopplung, die Rolle einer dedizierten Service-Fähigkeit bestehen. Der Controller nimmt seine Rolle nur dann ein, wenn eine seiner Fähigkeiten aufgerufen wird, die diese Rolle besitzt bzw. rechtfertigt. Besitzt der Service jedoch nur Fähigkeiten, deren Aufruf ihn als *Controller* qualifiziert, bezeichnet man ihn als *Designated Controller*.

[18] Vergleiche [Erl08, S.363 ff]
[19] Vergleiche [Erl08, S.387 ff]

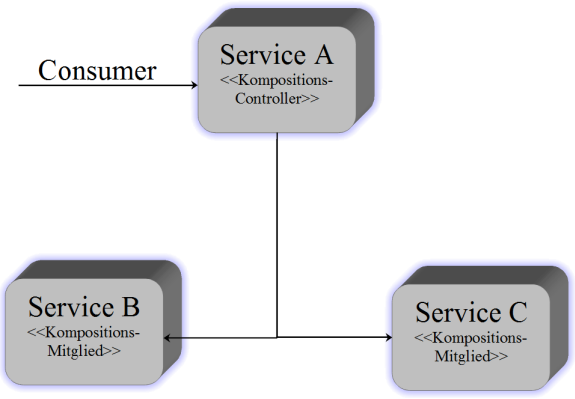

Abbildung 3.3: Service-Komposition

3.2.2 Vor- und Nachteile serviceorientierter Architekturen

Nach [Erl08] sind die am häufigsten angestrebten Ziele des serviceorientierten Computing:

- Verbesserung der inhärenten Interoperabilität
- Verbesserung der Föderation
- Verbesserung der Anbieter-Diversifizierung
- Verbesserung der Abstimmung von Geschäft und Technologie
- Verbesserung der Investitionsrentabilität (ROI)
- Verbesserung der Agilität der Organisation
- geringere IT-Belastung

Trotz dieser vielen Vorteile, wäre es ein Fehler zu denken SOA sei eine in jedem Fall anzustrebende Architektur. Vielmehr ist es eine Architektur für spezielle Anforderungen von heterogenen verteilten Systemen mit verschiedenen Eigentümern. Sind diese Voraussetzungen nicht gegeben können die Kosten für die Umsetzung der Architektur schnell den Nutzen übersteigen.[Jos08]

3.2.3 Serviceorientiertes Computing mit OSGi[20]

Zum überwiegenden Teil werden serviceorientierte Architekturen mit Hilfe von Web-Services, unter Nutzung von *HTTP* und *XML* als Kommunikationsplattform, realisiert. Vor allem im Zusammenhang mit der Anwendungsentwicklung auf Basis der *Eclipse*-Plattform bietet sich jedoch eine weitere Möglichkeit der Konzeption derartiger Architekturen. Die *Eclipse Rich-Client-Platform (RCP)* basiert seit der Version 3.0 auf *Equinox*, einem der derzeit fünf zertifizierten OSGi-konformen Produkte aus der Open-Source-Welt. Vor allem aufgrund des Shipments mit *Eclipse* ist *Equinox* derzeit die populärste OSGi-Implementierung.

Definition 3.2.4. *„The OSGi Service Platform specification delivers an open, common architecture for service providers, developers, software vendors, gateway operators and equipment vendors to develop, deploy and manage services in a coordinated fashion. It enables an entirely new category of smart devices due to its flexible and managed deployment of services.“[OA a, S.7]*

Abbildung 3.4 zeigt die grundlegende Architektur der OSGi-Service-Plattform. Sie

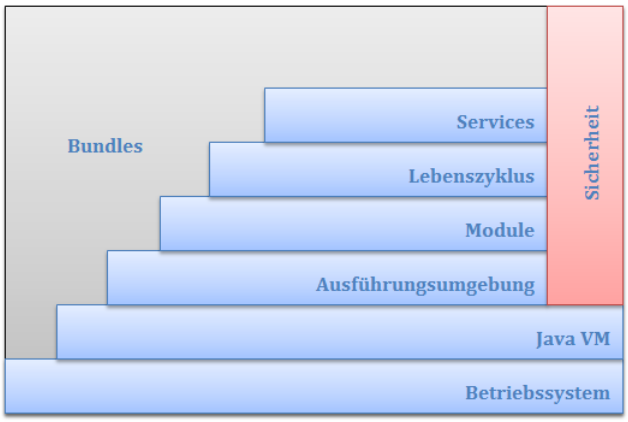

Abbildung 3.4: OSGi Schichtenmodell[21]

stellt also eine Laufzeitumgebung für Softwarekomponenten, welche auch als *Bundles* bezeichnet werden, innerhalb einer JVM dar. Ein solches *Bundle* reflektiert alle Facetten seiner eigenen Modularität in einem sogenannten *Bundle-Manifest*.

[20] vergleiche [OA b], [MVA10], [Wüt+08]
[21] in Anlehnung an [OA a, S.2]

Das beinhaltet vor allem Informationen zu seiner Identität, sowie seinen Fähigkeiten und Abhängigkeiten. Im Folgenden soll nach einem kurzen Überblick über die Lebenszyklus-Schicht, die von strategischer Bedeutung für das gesamte Framework ist, vor allem die Service-Schicht näher betrachtet und deren Ausrichtung an dem in den vorangegangenen Abschnitten diskutierten SOA-Paradigma beleuchtet werden.

Lebenszyklus-Schicht

Die Lebenszyklus-Schicht ermöglicht die Steuerung des Lebenszyklus eines *Bundles* mit Hilfe des *Bundle-Activators*. Dieser stellt gleichzeitig die Schnittstelle zum Framework dar und ermöglicht so den Zugriff auf tiefer liegende Schichten, insbesondere die *Service Registry*. Eine wohldefinierte *State-Machine* weist jedem *Bundle* einen von sechs Zuständen zu (siehe Tabelle 3.1).

INSTALLED	nach erfolgreicher Installation des *Bundles* auf der OSGi-Plattform
RESOLVED	alle Abhängigkeiten können aufgelöst werden, das *Bundle* kann gestartet werden oder wurde gestoppt
STARTING	*Bundle* wurde gestartet und befindet sich in der Ausführung der *start*-Methode, oder wartet bei aktivierter *Lazy-Activation-Policy* auf seine erste Verwendung
ACTIVE	*Bundle* ist aktiv
STOPPING	*stop*-Methode des *Bundle-Activators* wird abgearbeitet
UNINSTALLED	*Bundle* wurde deinstalliert

Tabelle 3.1: Übersicht Zustände von OSGi-Bundles

Service-Schicht

Die Kollaboration zwischen den einzelnen Bundles wird durch eine hochintegrierte Service-Schicht realisiert, deren Architektur sich an dem in Abschnitt 3.2.1 bespro-

chenen Modell orientiert. Jedes Bundle kann n Services unter n prinzipiell beliebigen Namen bei der *Service-Registry* anmelden. Hierfür wird der im *Bundle-Activator* übergebene *Bundle-Context* genutzt. In der Praxis hat sich die Verwendung des Namens des Service-Interfaces für die Registrierung durchgesetzt (Listing 3.1). Unter

```
public void start(BundleContext context) throws Exception{
   context.registerService(<Interface>.class.getName(), new <Interface>
      Impl(), null);
}
public void stop(BundleContext context) throws Exception{
   if(reference != null)
      context.ungetService(reference);
}
```

Listing 3.1: Register a Service (Auszug aus Activator)

Nutzung des selben Mechanismus kann natürlich auch ein Service bei der *Service-Registry* angefragt werden. Als Parameter nutzt man entweder den konkreten Namen unter dem ein Service registriert wurde, oder man verwendet Filter über die veröffentlichten Eigenschaften der Services, um alle Dienste zu finden, die definierten Anforderungen genügen. Im Ergebnis einer solchen Anfrage erhält man immer eine oder mehrere sogenannter *Service References*, welche alle Metadaten die einen bestimmten Service betreffen beinhalten. Nur unter Angabe dieser Referenz kann ein konkreter Service gebunden und in der Folge genutzt werden (Listing 3.2).

```
private <Interface> service;
private ServiceReference reference;

public void start(BundleContext context) throws Exception{
   ref = context.getServiceReference(<Interface>.class.getName());
   if(ref != null)
      service = (<Interface>)context.getService(ref);
}
public void stop(BundleContext context) throws Exception{
   if(reference != null)
      context.ungetService(reference);
}
```

Listing 3.2: Find a Service (Auszug aus Activator)

Bei der Registrierung kann außerdem optional ein java.util.Dictionary mit Properties hinzugefügt werden. Mit Hilfe dieser Zusatzangaben können Services gefiltert und

mit Laufzeitparametern versehen werden.
Abbildung 3.5 illustriert noch einmal das Zusammenspiel der einzelnen Schichten des OSGi-Frameworks.

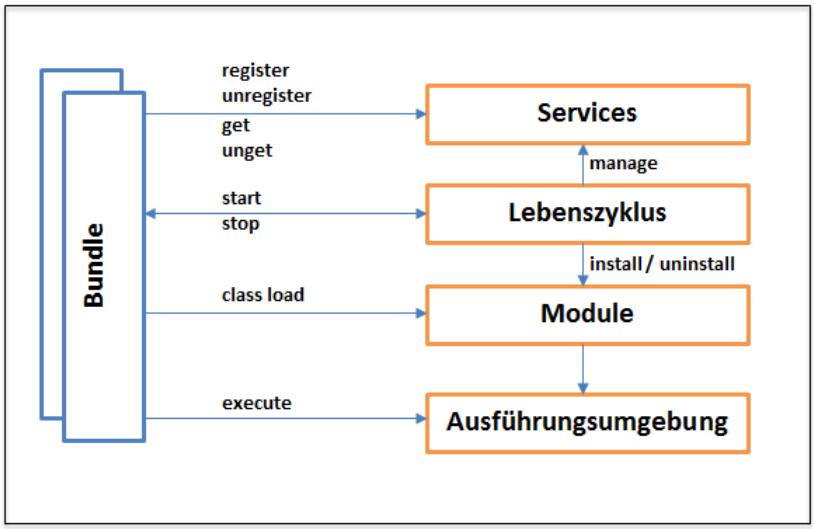

Abbildung 3.5: OSGi Interaction Model[22]

Die OSGI-Spezifikation legt eine Reihe von Standard-Services und Technologien für typische Anwendungsfälle fest, von denen eine willkürliche Auswahl in der nachfolgenden Liste wiedergegeben ist.[23]

- Tracker
- Declarative Services
- Remote Services
- Log Service
- Http Service
- Device Service
- User Admin Service

[22] in Anlehnung an [OA a, S.3]
[23] Eine vollständige Liste und die dazugehörigen Beschreibungen findet man in [OA a]

Im Folgenden soll nur auf die für die Erstellung SOA-basierter Anwendungen notwendigen Technologien, *Tracker*, *Declarative Services*, *Remote Services*, eingegangen werden.

Tracker Einer der Vorteile serviceorientierter Architekturen ist die Möglichkeit Dienste zur Laufzeit zur Anwendung hinzuzufügen oder zu entfernen. Um diesem dynamischen Aspekt innerhalb der Anwendungslogik Rechnung tragen zu können, sind Informationen über den Zeitpunkt der Verfügbarkeit determinierter Services erforderlich. Zum einen bietet hier die Plattform natürlich übliche Konzepte, welche auf dem *Observer-Pattern*[24] beruhen, an [OA a, Release 1], andererseits stellt sie hoch anpassbare Services zum Verfolgen (Tracking) von Service-Aktivitäten zur Verfügung [OA a, Release 2]. Ein Service-Tracker stellt Mechanismen zur Verfügung,

```
ServiceTracker tracker;

public void start(BundleContext context) throws Exception{
   tracker = new ServiceTracker(context, <Interface>.class.getName(),
      new MyServiceTrackerCustomizer()});
   tracker.open();
}
public void stop(BundleContext context) throws Exception{
   tracker.close();
}

class MyServiceTrackerCustomizer implements ServiceTrackerCustomizer{
   public Object addingService(ServiceReference ref){
      //do something useful if the service was added
   }
   public void modifiedService(ServiceReference ref, Object service){
      //do something useful if the service was modified
   }
   public void removedService(ServiceReference ref, Object service){
      //do something useful if the service was removed
   }
}
```

Listing 3.3: Call a Service Tracker

um auf Dynamik von Services, die der durch den Nutzer übergebenen Spezifikation entsprechen, geeignet reagieren zu können. Listing 3.3 illustriert das zugehörige

[24] Vergleiche [Gam+96]

Programmiermodell. Der optional zu implementierende *ServiceTrackerCustomizer* ermöglicht die Implementierung des benötigten Verhaltens beim Eintreten der standardisierten Service-bezogenen Ereignisse. Diese hohe Anpassbarkeit hat natürlich ihren Preis.

- Verlängerung der Startzeit in großen Systemen, da die Registrierung eines Services den Start des bereitstellenden Bundles und aller abhängigen Bundles erfordert.
- Ist ein Service registriert werden seine Implementierung und alle damit in Verbindung stehenden Klassen und Objekte in den Speicher geladen werden. Wird der Service nicht genutzt, ist dieser Speicherplatz unnötigerweise belegt.
- Die Komplexität steigt vor allem durch die aufwendige Implementierung von *ServiceTrackerCustomizern*, was zur Folge hat, dass die Wartbarkeit, Robustheit und Verlässlichkeit der Anwendung negativ beeinflusst werden.

Trotzdem ist der Einsatz des programmatischen Service-Models, insbesondere unter Verwendung von Trackern, vor allem in Systemen mit nur vereinzelten Services gut geeignet.

Declarative Services Um die im vorangegangenen Abschnitt aufgeworfenen Probleme zu vermeiden verabschiedete die OSGi-Alliance in ihrem Release 4 einen weiteren Ansatz, *Declarative Services*. Das zentrale Konzept bilden sogenannte *Service Components*, die ihre bereitgestellten und konsumierten Services deklarativ beschreiben. Eine solche Komponente besteht aus einer Komponentenklasse und einer Komponentenbeschreibung. Diese Beschreibung entspricht in vielerlei Hinsicht einem Service-Vertrag, wie er in Abschnitt 3.2.1 diskutiert wurde. Es handelt sich formal um ein XML-Dokument, was neben der Beschreibung der Komponente auch deren Abhängigkeiten beinhaltet. Die Komponentenklasse, eine einfache Java-Klasse, kann optional Callback-Methoden implementieren, um auf Lebenszyklus-Events zu reagieren.
Die zugehörige Laufzeitumgebung, *Service Component Runtime (SCR)*, implementiert die eigentliche OSGi-Spezifikation für Declarative Services und ist für die Steuerung des Lebenszyklus einschließlich der Instanzierung und Aktivierung einer *Service*

Component verantwortlich. Das deklarative Modell eröffnet gleichzeitig die Möglichkeit der verzögerten Aktivierung von Services (*Delayed Components*) gegenüber der sofortigen Instanzierung (*Immediate Components*). Durch Parsen des Service-Vertrages (Komponentenbeschreibung) ist die SCR in der Lage den Service bei der Service-Registry anzumelden, ihn aber erst in dem Moment zu instanzieren, wenn er das erste Mal angefragt wird. Ein weiterer signifikanter Vorteil ist die Möglichkeit der Auflösung von Service-Referenzen. Die SCR ist in der Lage Referenzen zu benötigten Services einer Service-Komposition aufzulösen und die Instanzierung des Services bis zur Befriedigung aller Abhängigkeiten zu verzögern.

Zur Nutzung deklarativer Services gibt es zwei Strategien:

Event-Strategie: Hierfür müssen in der Komponentenklasse zwei Methoden mit prinzipiell beliebigen Namen definiert werden, deren Rückgabewert *void* ist und deren einziger Parameter eine *ServiceReference* oder der Typ des Services, also das *Service-Interface* ist. Mit Hilfe dieser Methoden wird der Service von der SCR direkt in der Komponentenklasse gesetzt bzw. entfernt.

Lookup-Strategie: Im Gegensatz zur Event-Strategie wird hier die SCR nicht selbst tätig, sondern der Service wird über den *ComponentContext* per *locateService()* angefragt. Damit kann also die Instanzierung eines Services noch weiter verzögert werden, bis zum Zeitpunkt der konkreten Nutzung.

Listing 3.4 zeigt beispielhaft die Struktur einer Komponentenbeschreibung für sofortige Instanzierung nach der *Event-Strategie*. Soll die *Lookup-Strategie* verwendet werden, entfallen einfach die Attribute *bind* und *unbind*. Für jeden referenzierten

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="myComponent" immediate="true">
   <implementation class="TestComponent"/>
   <reference name="testService"
              interface="de.test.TestService"
              unbind="unbindTestService"
              bind="bindTestService"
              cardinality="1..1"
              policy="dynamic" />
</component>
```

Listing 3.4: Komponentenbeschreibung eines deklarativen Services

Service kann eine Kardinalität angegeben werden. Diese entscheidet, ähnlich wie in der UML, über Optionalität und Multiplizität. Im Beispiel in Listing 3.4 ist die Kardinalität auf *mandatory* und *unary* gesetzt, was bedeutet, dass die Komponente ohne referenzierten Service nicht instanziert werden kann und mit maximal einer Service-Referenz arbeitet.
Eine *Service Component* muss weiterhin definieren, wie mit der Dynamik in einer serviceorientierten Landschaft umgegangen werden soll. Hierfür wird das Attribut *policy* definiert. Es werden zwei Werte für dieses Attribut unterstützt, *dynamic* und *static*. Im ersten Fall bleibt die Komponente auch dann aktiv, wenn ein gebundener Service nicht mehr verfügbar ist, solange weiterhin alle Service-Referenzen erfüllt werden können. Im zweiten Fall wird die Komponente deaktiviert, wenn ein gebundener Service deregistriert wurde. Sie kann danach wieder aktiviert werden, wenn derselbe Service wieder registriert wird oder ein anderer die Referenz erfüllen kann. Muss ein Service parametrisiert werden, gibt es die Möglichkeit statt desselben eine sogenannte *ServiceFactory* instanzieren zu lassen. Nach Übergabe einer assoziativen Liste mit den Parametern erzeugt diese direkt den gewünschten Service. Als Voraussetzung muss in der Komponentenbeschreibung des entsprechenden Service Providers das Attribut *factory* gesetzt werden und auf den Namen der implementierten *ServiceFactory* verweisen. Der *Service Consumer* muss über das Attribut *target* einen Filter für den gesuchten Factory-Namen setzen.

Remote Services Die bisher vorgestellten Möglichkeiten zum Aufbau einer *SOA* mit Eclipse sind auf den Kontext einer JVM beschränkt. Um diese Grenze zu überschreiten, bedarf es eines weiteren Konzeptes, das der Remote Services, wie es in [OA b, Kapitel 13] definiert ist. Remote Services erlauben es Informationen über Distanz, über die Grenzen einer JVM hinweg, auszutauschen. Diese Fähigkeit ist in den Standard-Distributionen der OSGi-Frameworks nicht enthalten. Es gibt derzeit zwei Open-Source-Projekte, die entsprechende Funktionalitäten implementieren. Zum einen existiert das *Apache CXF Distributed OSGi*, welches gleichzeitig die Referenzimplementierung darstellt. Hier werden ganz klassisch Webservices auf Basis von *SOAP* und *WSDL* oder *RESTful JAX-RS* angeboten und konsumiert. Andererseits gibt es *r-OSGi* innerhalb des *Eclipse Communication Framework (ECF)*, welches im vorliegenden Fall aufgrund seiner Nähe zu *Equinox* von besonderem Interesse ist.

Allgemein beschreibt [OA b] die Architektur von Remote Services, wie in Abbildung 3.6. Ein sogenannter *DistributionProvider* nutzt die bestehende lokale serviceori-

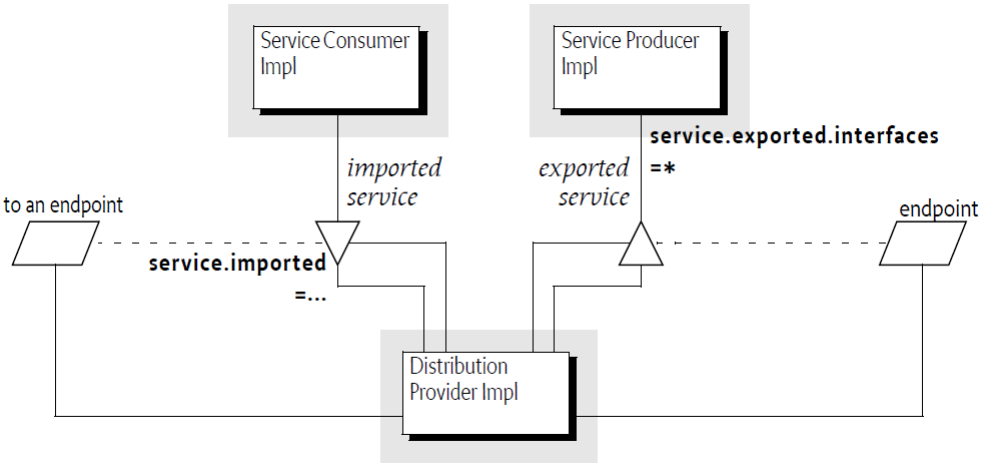

Abbildung 3.6: Architektur von Remote Services[25]

entierte Architektur und insbesondere die lose Kopplung zwischen den *Bundles* um registrierte Services zu exportieren, indem er einen Endpunkt erzeugt. Ein Endpunkt repräsentiert in diesem Zusammenhang den Zugriff auf einen Kommunikationsmechanismus zu einem Service in einem anderen Framework, einem Web-Service, einen anderen Prozess, etc. der ein Protokoll für die Kommunikation benötigt. Ein weiterer wichtiger Punkt ist die Möglichkeit der Nutzung von *Distributed Services* von anderen Plattformen und mit anderen Sprachen (beispielsweise C++ Client), aber auch die Nutzung von Services aus einem Non-OSGi-Umfeld in einem OSGi-Umfeld. Vor allem diesem Umstand ist es zu verdanken, dass mit OSGi vollwertige SOA-Landschaften aufgebaut werden können, bzw. eine OSGi-basierende Anwendung in eine solche nahtlos integriert werden kann.
Dieser *DistributionProvider* ist im *ECF* enthalten und mit verschiedensten Protokollen bestückbar. Wie im Eclipse-Umfeld üblich können fehlende Protokolle durch Eigenimplementierung per *Extension Point* hinzugefügt werden.
Um einen bestehenden deklarativen Service remote verfügbar zu machen sind prinzipiell nur Schritte notwendig:

1. Definition der *osgi.remote.interfaces* Variable. Als Werte werden die Namen

[25] in Anlehnung an [OA b, S.5]

der Interfaces hinzugefügt, die remote verfügbar gemacht werden sollen. Hier sind auch Wildcards, wie “*“ erlaubt.

2. Im ECF repräsentiert ein Container einen Endpunkt. Wird der *IContainerManager*, der dem *DistributionProvider* entspricht, bei der *ServiceRegistry* angemeldet, ist der richtige Zeitpunkt zum Erzeugen des Endpunktes und damit für die Veröffentlichung des/der Services gekommen. Listing 3.5 demonstriert, wie ein Endpunkt erzeugt wird.

```
public void start(final BundleContext context) throws Exception
{
   containerManagerTracker = new ServiceTracker(context,
      IContainerManager.class.getName(),
      new ServiceTrackerCustomizer(){
         public void removedService(ServiceReference reference,
                              Object service){
            if (container != null)
            container.disconnect();

            container = null;
         }
         public void modifiedService(ServiceReference reference,
                              Object service){}

         public Object addingService(ServiceReference reference){
            try{
               IContainerManager cM = (IContainerManager) context.
                  getService(reference);
               container = cM.getContainerFactory().createContainer("
                  ecf.r_osgi.peer");
               return containerManager;
            }catch (ContainerCreateException e){
               //Do some useful
            }
         }
   });
   containerManagerTracker.open();
}
```

Listing 3.5: Register a Remote Service

Das diese Überführung so einfach ist war bereits eine frühe Anforderung der OSGi-Alliance an Remote Services (Distributed OSGi).

„The solution is intended to allow a minimal set of distributed com-

> puting functionality to be used by OSGi developers without having to learn additional APIs and concepts. In other words, if developers are familiar with the OSGi programming model then they should be able to use ... this solution very naturally and straight forwardly to configure a distribution software solution into an OSGi environment."[OA c, S.5]

Aus diesem Grund sollen für den zu entwickelnden Prototyp vorerst deklarative Service eingesetzt werden, die später, bei konkretem Bedarf leicht remote verfügbar gemacht werden können.

4 Enterprise Integration Framework

Nicht mit Erfindungen,
sondern mit Verbesserungen
macht man Vermögen.

(Henry Ford)

Integration ist mehr als nur das Verketten von Prozessen oder das Injizieren von Daten eines Systems in ein anderes. Es ist vielmehr die Verbindung heterogener IT zu einem System neuer Qualität. Dieses Kapitel eröffnet mit einer kritischen Untersuchung der bisherigen Architektur der in [Spr09] entwickelten *Workflow-Engine*. Aus den Ergebnissen dieser Betrachtungen werden schrittweise Verbesserungen abgeleitet, die das System auf den Einsatz in einer verteilten Umgebung vorbereiten, Schwächen der bisherigen Architektur beseitigen und vor allem die Integration von Werkzeugen auf bidirektionaler Ebene ermöglichen. Abschließend wird das neue EIF-Architekturmodell vorgestellt.

4.1 Kritik am Konzept der Workflow-Engine

Die Architektur der *Remarc*® *Workflow-Engine*, wie in [Spr09] entwickelt, orientierte sich weitgehend an einem strikten Zwei-Schicht-Modell. Neben der eigentlichen Anwendungsschicht existierte eine einfache Persistenz-Schicht zur Anbindung des Datenmodells an eine Datenbank oder das Dateisystem, wie in Abbildung 4.1 illustriert. Die Anwendungsschicht beinhaltet neben der Schnittstelle zu den zu integrierenden Systemen insbesondere die Komponenten zum Editieren, Interpretieren und Ausführen von *Workflows*.
Die Weiterentwicklung des Prototyps und erste Versuche für einen praktischen Einsatz deckten recht schnell die Schwächen der bisherigen Architektur auf.

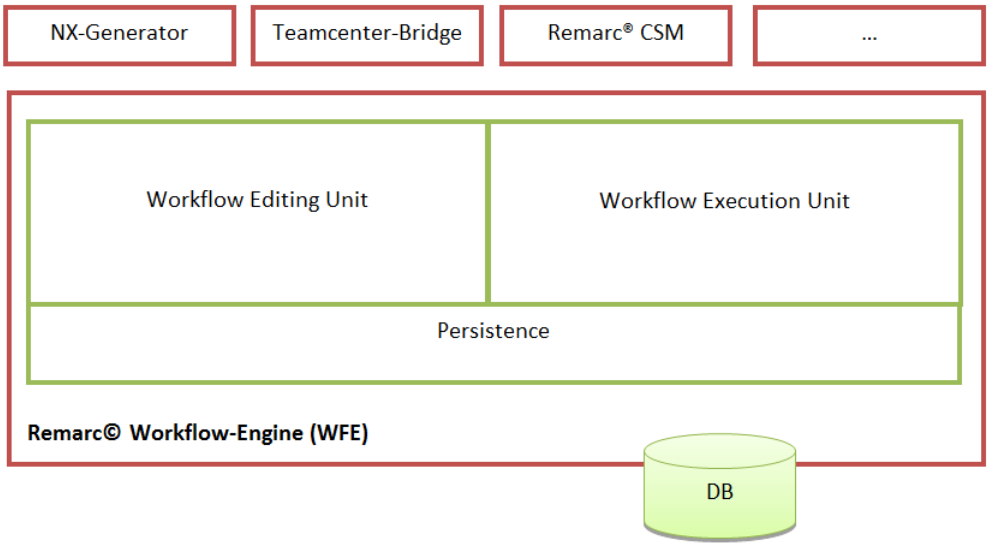

Abbildung 4.1: Workflow-Engine Schichtenmodell

4.1.1 Zwei-Schicht-Architektur

Die Zwei-Schicht-Architektur erschwert den kollaborativen Einsatz des Frameworks. Vorausgesetzt das System arbeitet mit einer zentralen Datenbank, greift jeder Client nur auf einen Snapshot der aktuellen Datenbasis zu. Werden zur gleichen Zeit an einem anderen Client Änderungen an den Daten vorgenommen, so erfahren die anderen davon vorerst nichts und arbeiten weiter mit den mittlerweile obsoleten Daten. Das Problem verschärft sich noch bei konkurrierenden Schreibzugriffen, welche auch unter dem Begriff *Lost Update* bekannt sind.

4.1.2 Service-Konzept

Die *ToolRegistry* wurde als *OSGi-Service* implementiert. Die Service-Nutzung erfolgte unter Verwendung eines *ServiceTrackers*. Die exzessive Verwendung dieser Technologie, auch bei der Weiterentwicklung von Remarc® CSM, führte zu stark verlängerten Startzeiten und einer starken Erhöhung der Komplexität.

4.1.3 GMF

Bereits bei der Implementierung des grafischen Editors des Prototyps der *Workflow-Engine* stellte sich heraus, dass das verwendete *GMF* das Customizing nur unzureichend unterstützt. Bei der Weiterentwicklung musste außerdem festgestellt werden, dass jede Änderung am Datenmodell einen hohen Aufwand bei der Anpassung des Editors nach sich zieht. Dies wird vor allem durch die Notwendigkeit, den gesamten modellgetriebenen Entwicklungszyklus durchlaufen zu müssen, verursacht.

4.1.4 Konfiguration und Parametrisierung

Das Mapping hochkomplexer Datenmodelle, wie es bei der bidirektionalen Integration von Portalen, wie *Siemens Teamcenter* vorkommt, ist bisher äußerst umständlich in einem Zusatz-Tab des Mapping-Editors realisiert. Hier wird mit Hilfe von komplizierten Schlüsselwerten versucht das Elemente des einen Datenmodells in Elemente eines anderen Modells zu überführen. Das bisherige Konzept ist äußerst fehleranfällig, benutzerunfreundlich und unflexibel.

4.2 Workflow-Engine wird EIF

Die im vorangegangenen Abschnitt diskutierten konzeptionellen Mängel wurden durch nachfolgend beschriebene Überarbeitungen und Erweiterungen der Architektur behoben. Insbesondere die Änderungen in den Abschnitten 4.2.2, 4.2.3 und 4.2.5 haben das System nachhaltig verändert und den Reifeprozess als Framework für Unternehmensanwendungen vorangetrieben. Aus dem einstigen Konzept *Remarc® Workflow Engine* wurde das *Remarc® Enterprise Integration Framework (EIF).*

4.2.1 Konzeptionelle Neuerungen am Datenmodell

Das Refactoring des bestehenden Datenmodells ist unumgänglicher Bestandteil des Reifeprozesses des Ziel-Frameworks. Nachfolgend beschriebene signifikante Ände-

rungen bilden die Basis aller weiteren Entwicklungsschritte. Abbildung 4.3 zeigt das neue Datenmodell in einer vereinfachten Darstellung.

Einführung von WorkflowElement

Die bisherige Lösung alle Elemente innerhalb eines *Workflows* und diesen selbst als Werkzeug (*Tool*) zu betrachten ist besonders im Hinblick auf die bidirektionale Integration nicht mehr ausreichend bzw. zu starr für die Aufnahme neuer Konzepte. Daher wurde das Interface *ITool* ersetzt durch das Interface *WorkflowElement*, welches alle gemeinsamen Operationen von Elementen eines *Workflows* im Allgemeinen deklariert.

Definition 4.2.1 (Workflow)**.** *Ein Workflow bildet einen mehrstufigen Prozess im Rahmen der Datenintegration als Modell ab.*

Definition 4.2.2 (WorkflowElement)**.** *Ein WorkflowElement ist jedes Element, dass Bestandteil eines* Workflows *sein kann, unabhängig vom technischen Ablauf und seiner interpretierten Bedeutung.*

Das bedeutet, dass ein Werkzeug (*Tool*) genauso ein *WorkflowElement* darstellt, wie beispielsweise eine Notiz, welche ein *Workflow* beinhalten kann. Jeder *Workflow* ist jedoch per Defintion ein Werkzeug (*Tool*), welches wiederum ein *WorkflowElement* darstellt. Das ursprünglich verwendete Kompositionsmuster bleibt demnach erhalten. Abbildung 4.3 verdeutlicht diesen Zusammenhang.

Klassifikation von Tools

Der ersatzlose Wegfall der im Datenmodell verankerten Klassifikation der Werkzeuge in die Klassen *Importer*, *Exporter*, *Validator*, *Filter* und *Messagehandler* erhöht die Flexibilität des Frameworks, da er das Hinzufügen neuer Werkzeugklassen ohne Veränderung des Datenmodells erlaubt. Die Klassifizierung, die in erster Linie der übersichtlichen Darstellung in der Werkzeugpalette des grafischen Editors dient, wird zukünftig durch einen weiteren Extension Point *Workflow Element Category* realisiert. Somit kann jedes beliebige Plugin Klassen (categories) definieren. Jedes

Workflow-Element kann sich für beliebig viele derartige Klassen durch Nennung der Klassen-ID registrieren. Der Workflow-Editor ist dann in der Lage die einzelnen *Workflow*-Elemente in die entsprechenden Kategorien einzusortieren.

WorkflowConnectionBar

Als ein nicht unwesentliches Problem hat sich herausgestellt, dass die Definition von *Workflows* zu nah am Teileerzeugungsprozess in Remarc® angelehnt war, so dass nachfolgend dargestellte Problematik nicht in den Vordergrund getreten ist. Ein Workflow kann auf zwei Wegen zum Einsatz kommen:

1. Er wird eigenständig zum Ablauf gebracht.
2. Er wird in einem anderen *Workflow* als *Tool* verbaut.

Die Daten, welche in den *Workflow* hineinfließen, kommen in beiden Fällen aus unterschiedlichen Quellen. Im ersten Fall muss sich das erste Werkzeug selbst um die Beschaffung und das letzte Werkzeug um die Ablage der Daten kümmern. Im anderen Fall müssen die Daten aus dem Werkzeug kommen, welches dem verbauten *Workflow* vorgeschaltet ist bzw. an das Werkzeug weitergegeben werden, welches dem *Workflow* nachgeschaltet ist. Bisher wurden beide Fälle gleich behandelt, d.h. die Daten wurden immer nach dem Schema für den ersten Fall beschafft bzw. abgelegt, was beim rekursiven Verbauen zu einem Datenbruch führt, den der Scheduler überbrücken muss (Abbildung 4.2).

Die Einführung eines Elementes zur Symbolisierung und Konfiguration der *Work-*

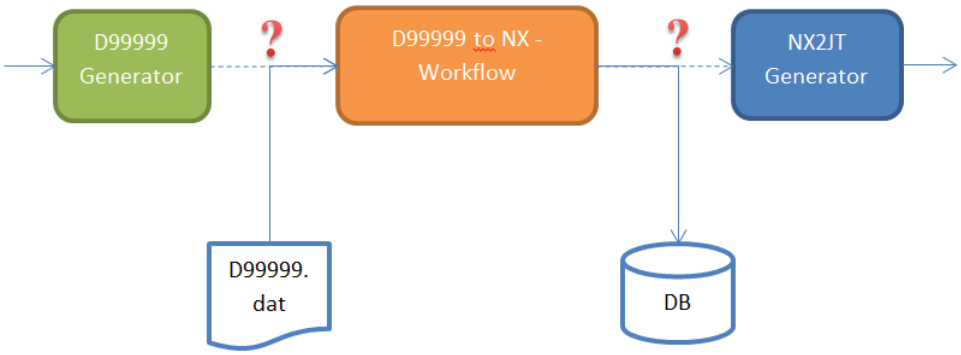

Abbildung 4.2: Problem der wechselnden Datenquellen und -senken

flow-Schnittstellen (*WorkflowConnectionBar*) soll Determiniertheit in diesen Prozess bringen.

Definition 4.2.3 (WorkflowConnectionBar)**.** *Eine* WorkflowConnectionBar *ist ein* WorkflowElement*, welches die Schnittstelle eines* Workflows *nach außen symbolisiert und beschreibt welche Daten in diesen hinein- und herausfließen.*

Wird der *Workflow* eigenständig zum Ablauf gebracht, ignoriert der Scheduler diese Elemente. Bei einem verbauten Workflow bezieht der Scheduler über die *WorkflowConnectionBar* Informationen über die Art der geforderten bzw. herausgegebenen Daten.

Datenquellen und -senken

Das neu eingeführte Konzept der Datenquellen und -senken, soll dem bidirektionalen Kommunikationsprozess Rechnung tragen. Diese abstrakten Konstrukte, welche ebenfalls WorkflowElemente verkörpern, sollen es ermöglichen Daten aus prinzipiell beliebigen Quellen zu erschließen bzw. Daten in prinzipiell beliebige Senken zu schieben.

Definition 4.2.4 (DataSource)**.** *DataSource ist ein WorkflowElement, das Daten von einer determinierten Quelle bezieht und über die Funktion getData() der Anwendung zur Verfügung stellt.*

Definition 4.2.5 (DataTarget)**.** *DataTarget ist ein WorkflowElement, das Daten an einer determinierten Stelle über die Funktion pushData() ablegt.*

Dabei möglicherweise auftretende Überschneidungen zum Werkzeug-Konzept können als ein positiver Nebeneffekt betrachtet werden, der die Usability und Anpassbarkeit des Systems erhöht, indem mehr als ein Weg zur Problemlösung zugelassen wird. So könnte beispielsweise der Anwendungsfall »Daten in Teamcenter importieren« durch ein Werkzeug unter Erweiterung von *Tool* oder alternativ als Datensenke unter Erweiterung von *DataTarget* realisiert werden. Des weiteren kann eine physische Anwendung natürlich beliebig viele Werkzeuge oder Datenquellen

bzw. Datensenken definieren. Somit kann dieselbe Anwendung kontextbezogen unterschiedlichen Anforderungen gerecht werden. Betrachtet am Beispiel von *Teamcenter* bedeutet das, dass die physische Anwendung an einer Stelle des Workflows als Datenquelle und an einer anderen als Datensenke auftreten kann.

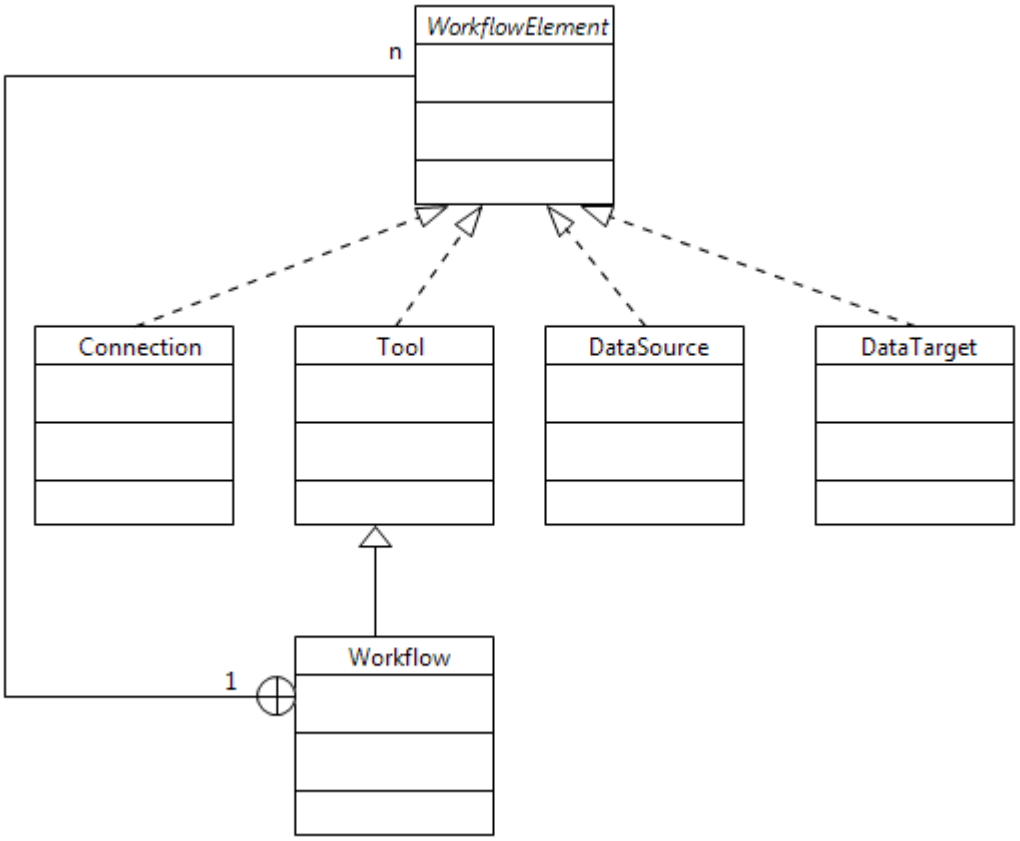

Abbildung 4.3: EIF-Datenmodell (vereinfacht)

4.2.2 Workflows im unternehmensweiten Zugriff

Einmal definierte Workflows, aber auch Konfigurationen müssen zukünftig unternehmensweit verfügbar gemacht werden, um Redundanzen zu vermeiden und Zeit sowie Kosten einzusparen. Außerdem muss nicht jeder Konstrukteur (Nutzer der integrierten Anwendung) geschult sein im Umgang mit dem neuen Framework. Vielmehr

sollte es Ziel sein rollenbasiert einzelne Nutzer zu Administratoren auszubilden, die in der Lage sind auch komplexe Workflows zu definieren und zu konfigurieren. Alle anderen Anwender merken von dem zugrunde liegenden Framework nichts oder wählen allenfalls anhand des Namens und einer Beschreibung den korrekten Workflow aus und starten die Verarbeitung.
Diese globale Verfügbarkeit soll mit der Implementierung eines CDO-Servers, wie in Kapitel 3.1 beschrieben, realisiert werden. Abbildung 4.4 zeigt das um den Server erweiterte Schichtenmodell. Die Erarbeitung der Architektur des Servers würde den

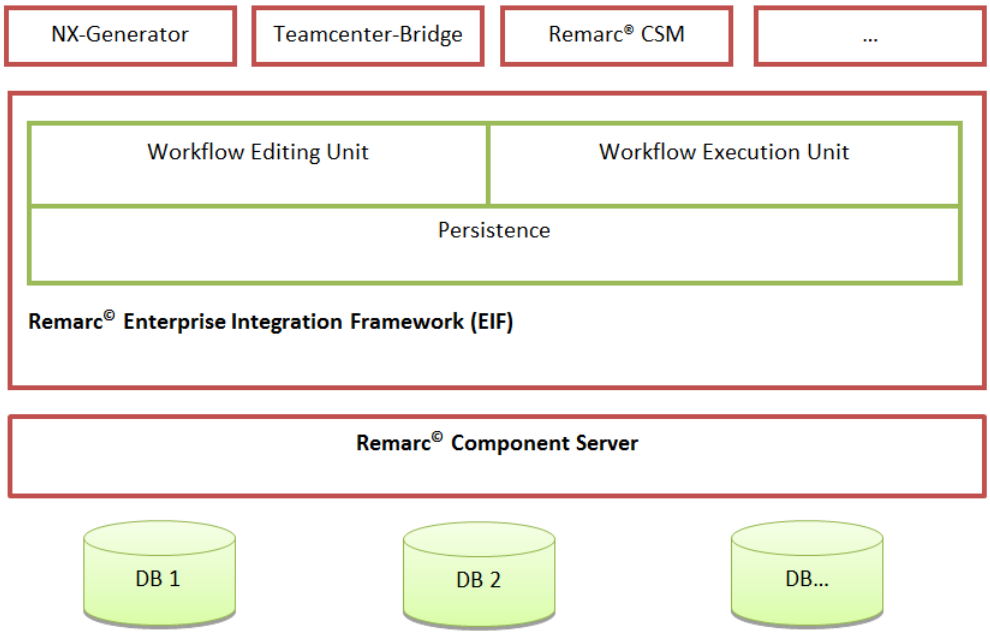

Abbildung 4.4: EIF-Schichtenmodell mit RCS

Rahmen dieser Arbeit überschreiten und bleibt deshalb einer weiteren vorbehalten. An dieser Stelle soll jedoch eine kurze Betrachtung der Client-Seite erfolgen, die im Schichtenmodell als Persistence-Schicht beschrieben ist. Diese Schicht beinhaltet im Wesentlichen einen sogenannten Model-Service, welcher deklarativ implementiert wird. Abschnitt 4.2.3 beschreibt die Erstellung eines deklarativen Services anhand der Workflow-Element-Registry. Aus diesem Grund soll hier nur auf Details des Service-Interfaces eingegangen werden.

Der Umgang mit dem EIF-Datenmodell und die persistente Ablage dieser Daten ist von hoher Komplexität. So sind beispielsweise Entscheidungen zu treffen, wel-

che Zugriffsrechte auf das jeweilige Modell-Element zu einem bestimmten Zeitpunkt erlaubt sind bzw. benötigt werden. Außerdem müssen Sitzungen bzw. Sitzungsdaten verwaltet werden, transaktionale Operationen auf dem Datenmodell ausgeführt und Oberflächen-Elemente benachrichtigt werden. Um den Zugriff auf die Persistenz-Mechanismen zu vereinfachen und diese Komplexität zu verbergen, soll das EIF-Model-Service-Interface als Fassade implementiert werden und so möglichst alle Low-level-Operationen kapseln. So muss einzig die High-Level-API im Service-Vertrag veröffentlicht werden.

Definition 4.2.6 (Fassade)**.** *„Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Benutzung des Subsystems vereinfacht.“[Gam+96, S.212]*

Alle Komponenten oder Prozesse (Threads), die Zugriff auf einen Teil des Datenmodells benötigen, müssen diese Ressource lokal teilen, um Transaktionalität auch auf dieser Ebene zu gewährleisten. Wird beispielsweise im Workflow-Editor eine Änderung durchgeführt, so muss sich diese zur selben Zeit auf alle offenen Views auswirken, unabhängig von einem *Commit* zur Datenbank (CDO-Server). Zu diesem Zweck wird jede Datenmodell-Ressource in einer sogenannten *EditingDomain* gespeichert, welche neben undo/redo-Funktionalitäten auch die korrekte Ausführung sämtlicher Operationen auf dem Datenmodell gewährleistet. Wird der Modell-Teilbaum von keiner Komponente mehr genutzt, muss die *EditingDomain* freigegeben werden. Würde bei der nächsten Anforderung die alte *EditingDomain* ausgegeben werden, könnte das zu einem inkonsistenten lokalen Datenmodell führen, vor allem dann, wenn zuvor nicht in die Datenbank übertragene (verworfene) Änderungen vorgenommen wurden. Deshalb muss sich jede Komponente für die Registrierung eines bestimmten Modellbereichs registrieren und bei Rückgabe der *EditingDomain* deregistrieren. Derartig registrierte Komponenten können ihre Änderungen speichern oder die Ausführung und Überwachung von Operationen auf dem Datenmodell veranlassen. Außerdem muss das Service-Interface Methoden zum holen und erzeugen von *Workflows* im Datenmodell bereitstellen, da das Kompositum das einzige Element ist, dass direkt auf dieser Ebene erzeugt werden kann und muss.
Neben diesen Basisoperationen muss das Interface noch um eine Reihe von Hilfsfunktionen erweitert werden, die beispielsweise die grafische Repräsentation eines *Workflows* mit diesem verknüpfen.

4.2.3 Workflow-Element-Registry als deklarativer Service

Mit der Erweiterung des Datenmodells wächst natürlich auch die Aufgabe die neu hinzugekommenen Elemente effektiv zu verwalten. Die bisher bestehende *ToolRegistry*, welche für die Verwaltung der einzelnen Werkzeugtypen verantwortlich war, muss nun dahingehend verändert werden, dass sie auch mit den neuen Workflow-Elementen umgehen kann. Da die statische Werkzeugtypisierung entfallen ist, muss auch die Suche und Filterung an das neue Konzept der dynamischen Typisierung angepasst werden. Dieser Wandel macht eine Umbenennung in *WorkflowElementRegistry* notwendig, da insbesondere nicht mehr nur Werkzeuge verwaltet werden. In diesem Zusammenhang soll außerdem die als OSGi-Service implementierten *ToolRegistry* zukünftig als *ServiceComponent* implementiert werden, um von den in Kapitel 3.2.3 diskutierten Vorteilen deklarativer Services profitieren zu können.

Definition der Extension Points Aufgabe der Workflow-Element-Registry ist die möglichst effektive Verwaltung aller in einem *Workflow* verwendbare Elemente. Deshalb ist dieses Bundle auch der korrekte Ort für die Definition der Extension Points für Kategorien und Workflow-Elemente. Der Extension Point für die Kategorien wird definiert als eine Sequenz von Kategorie-Namen des Typs String. Der Extension Point für Workflow-Elemente ist definiert als Sequenz von mindestens einem *Tool*, *DataSource* oder *DataTarget*, wobei jedes dieser Elemente einer bereits definierten Kategorie zugeordnet werden kann. Jedes Workflow-Element erhält daneben bei der Deklaration einen Namen, eine optionale Beschreibung und eine Factory-Klasse zugeordnet, die für die Erzeugung der konkreten Werkzeug-Instanz erforderlich ist.

Das Interface WorkflowElementFactory Für die Erzeugung der konkreten Werkzeuge soll das Muster Abstract Factory zum Einsatz kommen.

Definition 4.2.7 (Abstract Factory)**.** *„Biete eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen.“[Gam+96, S.107]*

Die im vorliegenden Fall von der abstrakten Fabrik, *WorkflowElementFactory*, zu erzeugende Produktfamilie besteht aus dem konkreten *WorkflowElement* und einer dazu passenden UI-Komponente zur Konfiguration des Elementes durch den Anwender. Der Vorteil liegt auf der Hand, der Client, beispielsweise das im nächsten Kapitel beschriebene Konfigurations-Framework, ist ausschließlich an die Schnittstellen der abstrakten Fabrik und der erzeugten Produkte der Produktfamilie gebunden. Die Typsicherheit wird durch Implementierung als generische Klasse sichergestellt. Listing 4.1 zeigt die wichtigsten Methoden dieser Schnittstelle.
Nachteilig bei dieser Implementierung ist, dass bei Erweiterung der Mitglieder der

```
public interface WorkflowElementFactory<T extends WorkflowElement> {

   T createWorkflowElement(String name);

   AbstractWorkflowElementComposite createWorkflowElementComposite(
         DataSource source, Composite parent, int style);

   boolean isFactoryFor(WorkflowElement element);
}
```

Listing 4.1: Interface WorkflowElementFactory

Produktfamilie relativ viel Klassen betroffen sind, die gleichfalls geändert werden müssen. Dennoch ist es eine elegante Möglichkeit Klienten mit unterschiedlichen Objektkonfigurationen zu versehen, wie es im vorliegenden Fall erforderlich ist.

Komponentenbeschreibung Erster Schritt bei der Implementierung eines deklarativen Services ist die Vereinbarung des Service-Vertrages, im OSGi-Kontext der Komponentenbeschreibung. Hier soll festgelegt werden, dass Funktionalitäten, die in einem Interface WorkflowElementRegistry vereinbart sind als Service von der Klasse WorkflowElementRegistryImpl bereitgestellt werden, sofern der im vorangegangenen Abschnitt definierte Dienst EIFModelService verfügbar ist. Es handelt sich also hier um eine Service-Komposition. Zur Bindung soll vorerst die Event-Strategie in Verbindung mit der dynamic-Policy genutzt werden, d.h. es muss eine entsprechende bind- und unbind-Methode definiert werden.

Service-Interface Das Service-Interface *WorkflowElementRegistry* beschreibt die angebotenen Dienste. Diese bestehen im Wesentlichen aus dem Zugriff auf alle definierten Kategorien und Workflow-Elemente. Eine Zentrale Rolle spielt eine Funktion zur strukturierten Suche von Workflow-Elementen anhand ihrer Merkmale. So kann beispielsweise nach Elementen einer bestimmten Kategorie oder eines bestimmten Typs gesucht werden. Zur Performance-Verbesserung, aber auch um nachträglich beispielsweise Oberflächenkomponenten erstellen zu können, werden die zur Erzeugung genutzten Factories in einer Liste gespeichert und können bei Bedarf mit Hilfe der Methode *isFactoryFor(WorkflowElement)* identifiziert und genutzt werden. Anhang B.1 zeigt die vollständige Interface-Definition.

4.2.4 Neuentwicklung des grafischen Editors[26]

Um die bereits erörterten Probleme bei der bisherigen Umsetzung des grafischen Editors mit GMF zu umgehen, wurde Graphiti, ein weiteres auf dem GEF basierendes Framework, auf seine Tauglichkeit zur Implementierung eines Workflow-Editors im vorliegenden Umfeld hin untersucht.
GEF ist ein hochkomplexes Framework zur Erstellung grafischer Editoren. Während GMF versucht diese Komplexität durch modellbasierte Code-Generierung beherrschbar zu machen, verfolgt Graphiti einen streng API-zentrierten Ansatz, indem es versucht, vollständig die zugrundeliegenden Basistechnologien aus Draw2D und GEF zu kapseln und dem Entwickler eine einfache, beherrschbare API zur Entwicklung homogener Editoren zur Verfügung zu stellen. Dabei bietet es bei der Entwicklung und im Einsatz insbesondere die folgenden Vorteile:

- schnelle Einarbeitung durch Kapselung der GEF- und Draw2D-API
- inkrementelle Entwicklung durch ausbaubare Standardimplementierungen
- homogene Editoren durch ähnliches Aussehen und Verhalten
- plattformunabhängige Diagramm-Definition

[26] vergleiche [Bra+09]

Abbildung 4.5 zeigt die grundlegende Architektur des Graphiti-Frameworks. Zentrale Komponente ist, wie man hier sieht, neben der Laufzeit-Komponente der *DiagramTypeAgent*, welcher vom Entwickler zu implementieren ist. Dieser ist für die Realisierung aller Benutzerinteraktionen, wie Create, Move, Resize,etc., und insbesondere für die Pflege der Modelldaten verantwortlich. Standardimplementierungen und Services sorgen für erste schnelle Erfolge in Form prototypischer Editoren, die in späteren Iterationen und mit wachsendem Know-How weiter ausgebaut werden können. Das *Domain Model* enthält die *Business Objects*, im vorliegenden Fall also das weiter oben beschriebene EIF-Datenmodell. Das *Pictogram Model* enthält alle Informationen zur Darstellung des Diagramms und seiner Elemente. Dieses Modell ist völlig unabhängig vom *Domain Model*. Es könnte also ein Diagramm auch ohne hinterlegte *Business Objects* darstellen. Das *Link Model* verknüpft die beiden Modelle und ermöglicht unter anderem die Synchronisation zwischen den beiden anderen Modellen. Während das *Domain Model* vom Entwickler zur Verfügung gestellt werden muss, wird das *Pictogram-* und *Link-Model* von Graphiti bereits mitgeliefert. Der funktionale Aufbau des *DiagramTypeAgent* ist streng Feature-basiert. Somit muss jede Operation als eigenes Feature implementiert werden, wobei auch hier für Basisoperationen Standardimplementierungen existieren.

Für die Umsetzung des Workflow-Editors muss Graphiti der Typ des Diagrammes, der zugehörige *DiagramTypeProvider* und ein *ImageProvider* per Extension Point bekannt gemacht werden. Zur Laufzeit kann das Framework dann anhand des Diagrammtyps entscheiden an welchen *DiagramTypeProvider* Events der *Interaction Component* weitergeleitet werden müssen und woher ggf. Bilder zur Darstellung bezogen werden können. Im nächsten Schritt muss für jedes WorkflowElement ein ADD-, CREATE- und DELETE-Feature implementiert werden. Entsprechendes gilt auch für die Bereitstellung von *Direct Editing Support* und *Copy-/Paste*-Funktionalitäten.

Dieser kurze Überblick soll ausreichend sein zu zeigen, dass mit Graphiti auf einfache Weise ein Editor erstellt werden kann, der den Anforderungen für den WorkflowEditor genügt und die Weiterentwicklung in sehr kurzen Iterationszyklen erlaubt, was insbesondere bei GMF nicht der Fall war.

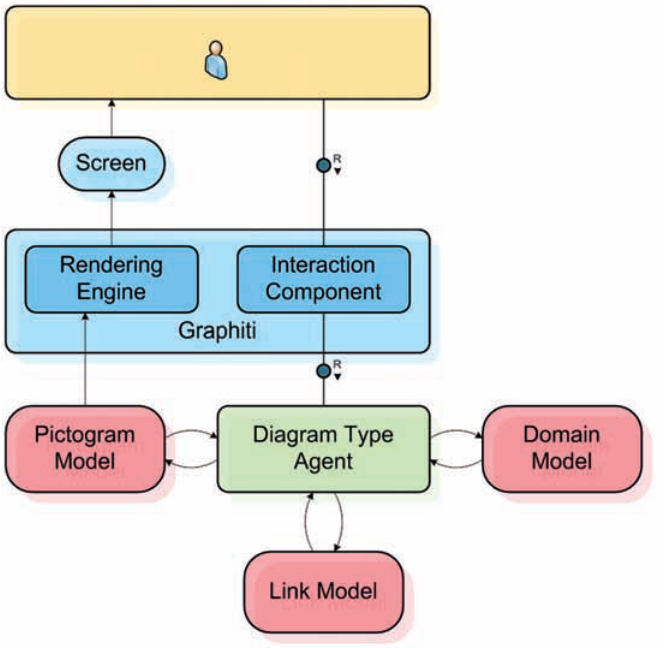

Abbildung 4.5: Graphiti Architektur-Modell[Bra+09, S.68]

4.2.5 Einführung einer Konfigurationsschicht

Bei der Integration von Softwaresystemen auf Datenebene ist die Konfiguration und Parametrisierung der einzelnen am Integrationsprozess beteiligten Werkzeuge von exponierter Bedeutung. Mit Hilfe dieser Parametrisierung werden die Integrationspartner gesteuert, Konventionen übermittelt und Response-Parameter übertragen. Neben der dynamischen oder regelbasierten Ermittlung einzelner Parameter sind vor allem Abhängigkeiten zwischen verschiedenen Parametern und deren Auflösung zur Laufzeit von Bedeutung. Diese hochkomplexe Aufgabe soll durch die Einführung einer zusätzlichen Konfigurationsschicht realisiert werden. Abbildung 4.6 zeigt das dahingehend erweiterte Schichtenmodell des EIF. Im Kapitel 5 wird die Architektur dieser Schicht ausführlich erörtert.

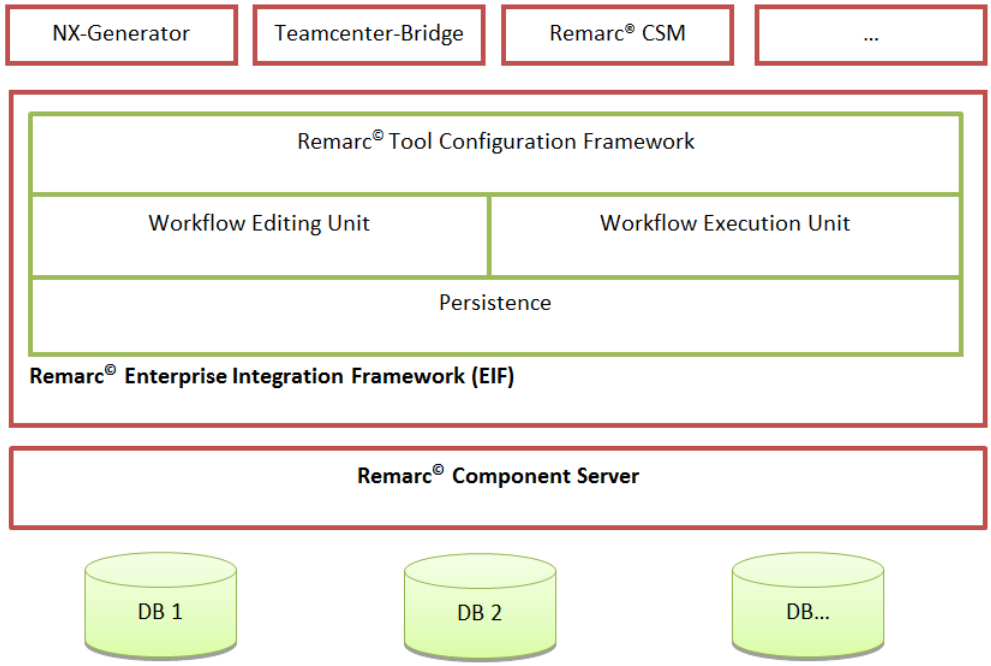

Abbildung 4.6: EIF-Schichtenmodell mit Tool Configuration Framework

5 EIF Konfigurations-Framework

Bei maximaler Variabilität aller Parameter wird jeder Traum zur Realität.

(Jörg Lohrmann, *1970)

5.1 Anforderungsanalyse

Im Rahmen eines Kickoff-Meetings wurden mit den Stakeholdern Anforderungen und ein gemeinsames Vokabular erarbeitet. Die Ergebnisse werden im Folgenden kurz vorgestellt.

- Bereitstellung und grafische Darstellung von Konfigurationsdaten aller mit Hilfe des EIF integrierten Software-Werkzeuge unter Nutzung von Konfigurationsbäumen. Jeder Komponente soll genau ein solcher Konfigurationsbaum zugeordnet werden.
- Konfigurationswerte sollen manuell eingegeben, aus Wertelisten unter Nutzung des Datenquellen-Konzeptes ausgewählt oder mit Hilfe von Skriptsprachen beschrieben werden können.
- Konfigurationsschlüssel sind in ihrer Hierarchie eindeutig und können frei definiert oder aus Wertelisten unter Nutzung des Datenquellen-Konzeptes ausgewählt werden.
- Abhängigkeiten zwischen Schlüsseln unterschiedlicher Hierarchiestufen müssen modellierbar sein, um beispielsweise mehrstufige Teamcenter-LOV's abbilden zu können.
- Konfigurationsgruppen können Zwangsbedingungen festlegen, welche die zugehörigen Konfigurationsschlüssel definieren müssen.

- Das entstehende Konfigurationsmodell muss sich in vorhandene zumindest EMF-basierte Datenmodelle integrieren lassen.
- Konfigurationen müssen, unternehmensweit abrufbar, als Muster persistierbar sein.

Im Rahmen dieser Veranstaltung wurde außerdem ein Domänen-Vokabular festgelegt, welches das Verständnis zwischen allen Beteiligten sicherstellen soll.

Definition 5.1.1 (Konfiguration)**.** *Eine Konfiguration ist ein Wald von Konfigurationsbäumen.*

Definition 5.1.2 (Konfigurationsbaum)**.** *Ein Konfigurationsbaum ist ein Baum im graphentheoretischen Sinn, dessen Wurzel durch eine Konfigurationsgruppe, die den Namen der zu konfigurierenden Softwarekomponente* (WorkflowElement) *trägt, gebildet wird.*

Definition 5.1.3 (Konfigurationsgruppe)**.** *Eine Konfigurationsgruppe ist jeder Knoten eines Konfigurationsbaumes. Sie fungiert als Container-Element zur Aufnahme von n Konfigurationsschlüsseln oder weiteren strukturierenden Konfigurationsgruppen und kann Zwangsbedingungen (Constraints) an ihre untergeordneten Elemente vererben.*

Definition 5.1.4 (Gruppen-Constraint)**.** *Ein Gruppen-Constraint ist eine Zwangsbedingung, die von einer Konfigurationsgruppe für ihre untergeordneten Elemente deklariert werden kann. Alle Konfigurationsschlüssel müssen diese Constraints definieren.*

Definition 5.1.5 (Konfigurationsschlüssel)**.** *Jedes Blatt eines Konfigurationsbaumes heißt Konfigurationsschlüssel. Ihm kann genau ein Wert zugewiesen werden.*

5.1.1 Definition der Anwendungsfälle

Nachdem die Anforderungen skizziert und das Vokabular festgelegt worden ist, sollen nun die Anwendungsfälle aus diesen Vorgaben spezifiziert und konkretisiert werden. Abbildung 5.1 stellt die Beziehungen zwischen den Akteuren und den einzelnen Anwendungsfällen dar und zeigt die Trennung zwischen System und Umwelt (Rechteck).

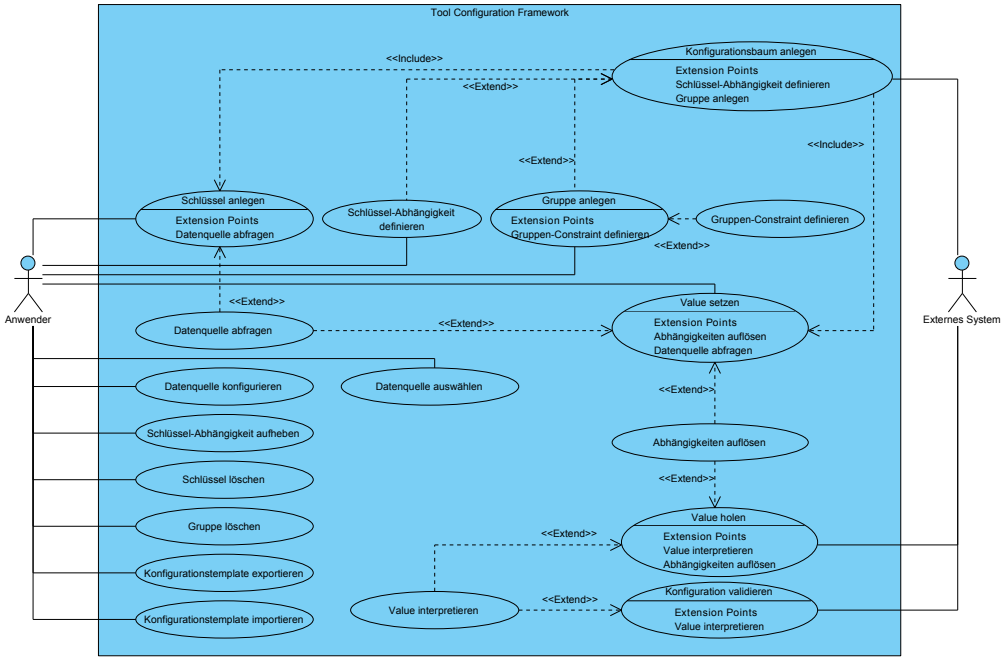

Abbildung 5.1: Use-Case-Diagram der Werkzeug-Konfiguration

a) Aktoren

Anwender: bezeichnet jeden menschlichen Akteur, der das System bedienen muss. Da es sich um ein offenes Framework handelt, ist diese Rolle nicht weiter eingeschränkt. Einschränkungen jeder Art müssen durch das Zielsystem erfolgen. Im Falle von Remarc® CSM erfolgt die Einschränkung durch Angliederung der Werkzeug-Konfiguration an die Rolle des *Creators*.

Externes System: bezeichnet jegliche Software, die am Integrationsprozess teilnehmen soll und die erforderlichen Schnittstellen des EIF implementiert.

b) Use Cases

Konfigurationsbaum anlegen: Ein zu integrierendes Werkzeug (*Externes System*) definiert einen Konfigurationsbaum mit dessen Hilfe es alle benötigten und unterstützten Konfigurationsgruppen und -schlüssel und deren Standard-Attributierung bereitstellt. Insbesondere beinhaltet dieser Anwendungsfall die Anwendungsfälle *Schlüssel anlegen* und *Value setzen*, und kann im Einzelfall durch die Anwendungsfälle *Schlüssel-Abhängigkeit definieren*, *Gruppe anlegen* und *Gruppen-Constraint definieren* erweitert werden.

Gruppe anlegen: Ein Anwender kann eine neue Gruppe an den von einem Werkzeug im Konfigurationsbaum definierten Stellen anlegen. Ein Externes System kann prinzipiell beliebige Gruppen anlegen, allerdings nur initial im Rahmen des Anwendungsfalls *Konfigurationsbaum anlegen*. Werden Gruppen-Constraints benötigt kann der Anwendungsfall *Gruppen-Constraint definieren* herangezogen werden.

Gruppe löschen: Ein Anwender kann eine von ihm erstellte Gruppe löschen. Initiale (durch ein externes System angelegte) Gruppen können von keinem Akteur mehr gelöscht werden.

Gruppen-Constraint definieren: Für jede Gruppe können von den Aktoren Constraints deklariert bzw. definiert werden. Unter Constraints sind in diesem Zusammenhang Attribute zu verstehen, die an jeden Schlüssel einer Gruppe vererbt werden und deren Belegung nicht optional ist. Derartige Zwangsbedingungen können von einem Aktor nur im Rahmen des Anwendungsfalls *Gruppe anlegen* erstellt werden.

Schlüssel anlegen: Ein Anwender kann einen neuen Schlüssel an den von einem externen System im Konfigurationsbaum definierten Stellen anlegen, wobei der erlaubte Schlüsselraum mit Hilfe des Anwendungsfalls *Datenquelle abfragen* bestimmt werden kann. Im Gegensatz dazu kann ein *Externes System* Schlüssel nur initial im Rahmen des Anwendungsfalls *Konfigurationsbaum anlegen* definieren. Ein neu angelegter Schlüssel kann als Pflichtelement gekennzeichnet werden. Soll bei der Schlüsselvergabe ein bestimmter Namensraum verwendet werden, kann der Anwendungsfall *Datenquelle abfragen* herangezogen werden.

Schlüssel löschen: Ein Anwender kann einen von ihm erstellten Schlüssel löschen. Vom externen System definierte Schlüssel können nicht gelöscht werden.

Schlüssel-Abhängigkeit definieren: Auswählbare Werte eines Schlüssels können vom aktuellen Wert eines anderen Schlüssels abhängig sein. Das erlaubt beispielsweise die Nachbildung von hierarchischen Wertelisten, wie sie in Teamcenter verwendet werden. Ein Anwender kann eine oder mehrere solcher Abhängigkeiten für jeden Schlüssel definieren. Eine automatische Zyklen-Erkennung sollte in einer späteren Ausbaustufe den Anwender bei der korrekten Definition von Abhängigkeiten unterstützen. Im Gegensatz dazu kann ein *Externes System* Schlüssel-Abhängigkeiten nur initial im Rahmen des Anwendungsfalls *Konfigurationsbaum anlegen* definieren.

Schlüssel-Abhängigkeit aufheben: Ein Anwender kann gesetzte Abhängigkeiten zu anderen Schlüsseln entfernen. Vom externen System definierte Schlüssel-Abhängigkeiten können nicht gelöscht werden.

Konfigurationstemplate exportieren: Eine Konfiguration kann durch den Anwender persistent als Vorlage abgelegt werden.

Konfigurationstemplate importieren: Eine als Template gespeicherte Konfiguration kann durch den Anwender importiert oder an eine bestehende Konfiguration angehängt werden, dabei werden eventuell bestehende gleichnamige Konfigurationsbäume ersetzt. Es ist möglich nur einzelne, explizit ausgewählte Konfigurationsbäume aus der gespeicherten Konfiguration zu importieren.

Datenquelle auswählen: Der Anwender kann für einen Schlüssel oder eine Gruppe eine vorhandene Datenquelle zur Definition des Wertebereichs festlegen.

Datenquelle konfigurieren: Der Anwender kann eine ausgewählte Datenquelle konfigurieren, sofern diese der Konfiguration bzw. Parametrisierung bedarf, um Quelldaten bereitstellen zu können.

Datenquelle abfragen: Ein Akteur erhält durch Abfrage einer konfigurierten Datenquelle eine Liste mit gültigen Werten für einen determinierten Schlüssel. Diese Abfragen erfolgen ausschließlich optional im Rahmen der Anwendungsfälle *Value setzen* und *Schlüssel anlegen*.

Value setzen: Ein Akteur kann einem Schlüssel einen konkreten Wert zuweisen. Diese Definition erfolgt manuell oder unter Verwendung des Anwendungsfalls *Datenquelle abfragen*. Sind Abhängigkeiten für den Schlüssel definiert, müssen diese durch den Anwendungsfall *Abhängigkeiten auflösen* gelöst werden.

Value holen: Ein *Externes System* kann den zugewiesenen Wert eines Schlüssels abfragen. Bei Bedarf kann dabei der Anwendungsfall *Value interpretieren* hinzugezogen werden, um vorhandene Skripte zu interpretieren. Müssen Abhängigkeiten zur Laufzeit des Workflows beachtet werden, können diese durch Aufruf des Anwendungsfall *Abhängigkeiten auflösen* gelöst werden.

Abhängigkeiten auflösen: Sind Abhängigkeiten zwischen einzelnen Schlüsseln definiert (siehe Anwendungsfall *Schlüssel-Abhängigkeit definieren*)worden, so müssen diese entweder Zur Editierzeit im Rahmen des Anwendungsfalls *Value setzen* oder zur Laufzeit des entsprechenden Workflows im Rahmen des Anwendungsfalls *Value holen* aufgelöst werden.

Konfiguration validieren: Ein externes System kann eine Konfiguration auf ihre strukturelle und syntaktische Korrektheit prüfen. Bei Bedarf können genutzte Skripte mit Hilfe des Anwendungsfalles *Value interpretieren* auf ihre Interpretierbarkeit (syntaktische Korrektheit) hin untersucht werden.

Value interpretieren: Werden Werte einzelner Schlüssel mit Hilfe eines Skriptsprache definiert, müssen diese zur Laufzeit eines Workflows interpretiert werden, um den aktuellen Wert zu ermitteln. Die Anwendungsfälle *Value holen* und *Konfiguration validieren* können bei Bedarf die Validierung initiieren.

5.1.2 Analysemodell

Das Analysemodell vermittelt einen ersten Eindruck vom zu entwerfenden Konfigurations-Framework und grenzt grob operative Komponenten ab, welche mit Hilfe der Anwendungsfälle (Use Cases) identifiziert wurden. Zur Darstellung wurde ein vereinfachtes Klassendiagramm (Abbildung 5.2) gewählt, da es mit seinen wenigen klaren Inhalten das Gesamtprojekt skizziert, ohne in diesem frühen Entwicklungsstadium zu sehr auf Details einzugehen und diese damit zu früh zu fixieren.

Data Model beinhaltet alle Konfigurationsdaten einer Softwarekomponente, deren Abhängigkeiten, Constraints und verwendete Datenquellen. Es handelt sich aus diesem Grund um eine echte Teilmenge des Datenmodells des Enterprise Integration Framework.

Configuration Navigator ermöglicht das Anlegen, Bearbeiten und Löschen von Elementen der Konfigurationsstruktur. Es ist die zentrale Komponente zur Navigation im Konfigurationsbaum und zur Auswahl einzelner Elemente.

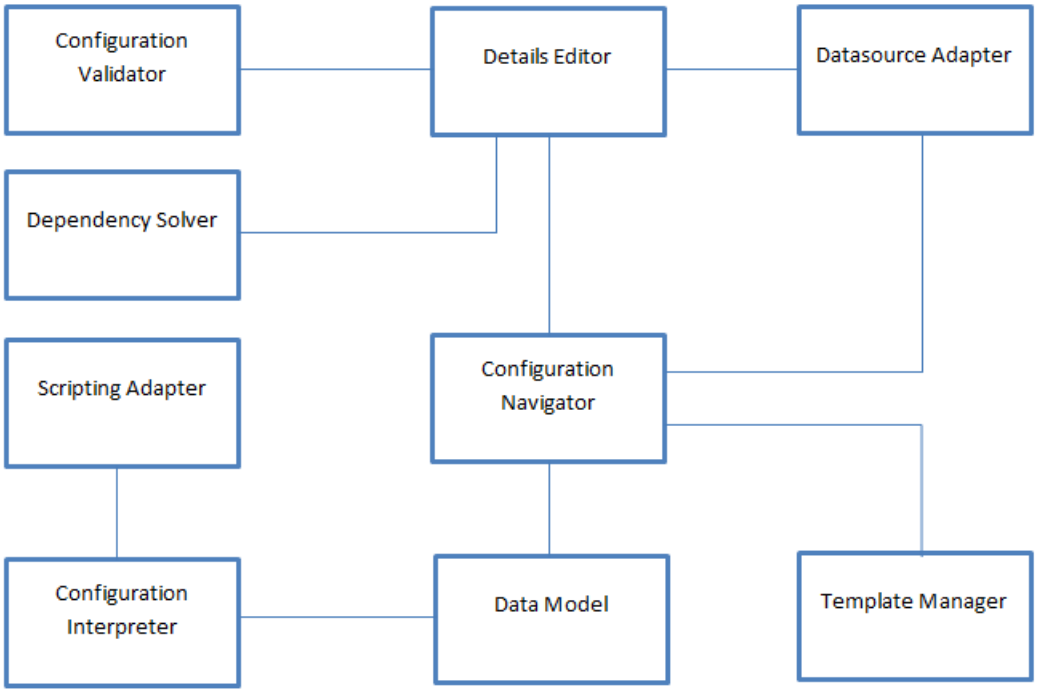

Abbildung 5.2: Analysemodell Configuration Framework

Details Editor erlaubt das Editieren der Attribute eines einzelnen Elements des Konfigurationsbaumes, insbesondere dessen Wert, Abhängigkeiten, Constraints und verwendete Datenquelle.

Scripting Adapter stellt verschiedene Skript-Interpreter zur Verfügung, mit deren Hilfe Konfigurationswerte berechnet oder sonst interpretiert werden können.

Datasource Adapter stellt verschiedene Datenquellen zur Verfügung, mit deren Hilfe Auswahldaten für Konfigurationswerte oder erlaubte Schlüsselwerte zur Verfügung gestellt werden können.

Configuration Validator prüft die Gültigkeit einer Konfiguration hinsichtlich der Erfüllung von Pflichten, Constraints und Abhängigkeiten, sowie die syntaktische Korrektheit von Skripten.

Dependency Solver löst Abhängigkeiten zwischen einzelnen Elementen des Konfigurationsbaumes auf und ermöglicht die Navigation zwischen diesen.

Configuration Interpreter interpretiert Konfigurationswerte unter Zuhilfenahme des Scripting Adapters und stellt sie unter Nutzung der Integrationsplattform der zugehörigen Softwarekomponente zur Verfügung.

Template Manager ermöglicht das Speichern, Laden und Anhängen einzelner Konfigurationsbäume zur unternehmensweiten Wiederverwendung.

5.2 Architekturerstellung

in diesem Kapitel werden die maßgeblichen Einflussfaktoren produktspezifischer, technologischer und organisatorischer Art ermittelt, von denen die nichtfunktionalen Anforderungen abgeleitet werden können. Anschließend soll beispielhaft eine Lösungsstrategie entwickelt werden, die unter anderem im letzten Abschnitt umgesetzt wird. Auf die abschließende Architekturbewertung musste aus Kostengründen vorerst verzichtet werden.

5.2.1 Spezifikation der Einflussfaktoren

Die Spezifikation von Einflussfaktoren hat maßgeblichen Einfluss auf die zu erstellende Architektur. Die Tabellen 5.1, 5.3, 5.4, 5.2 geben einen Überblick über die wichtigsten Faktoren.

5.2.2 Risiken und Lösungsstrategien

Nach der Analyse der Einflussfaktoren müssen die sich daraus ergebenden Risiken identifiziert werden. An dieser Stelle soll beispielhaft eines der ermittelten Risiken herausgegriffen und zur Entwicklung einer Lösungsstrategie herangezogen werden.

Integrierbarkeit und Austauschbarkeit der Benutzerschnittstelle

Die Benutzerschnittstelle soll einerseits leicht austauschbar sein, um in späteren Versionen Schnittstellen für verschiedenste Laufzeit-Umgebungen anbieten zu können,

Produktspezifische Faktoren	Flexibilität und Veränderbarkeit	Einfluss
P1: Änderbarkeit P1.1 Modularität Das Framework soll nach dem Baukastenprinzip an Kundenanforderungen und neue Technologien angepasst bzw. erweitert werden.	Die Anpassung des Systems an die kundenspezifische, eher heterogene IT-Landschaft ist Voraussetzung für dessen Einsatz. Ein komponentenbasiertes System sichert langfristigen Erfolg, da auf Kundenwünsche flexibel reagiert werden kann, ohne die Konsistenz der Systemarchitektur zu verletzen.	Design-Entscheidung
P2: Portierbarkeit *P2.1 Externe Systeme* Das Framework soll zukünftig auf weitere Plattformen portiert werden.	Um konkurrenzfähig zu bleiben, sollen auch Kunden mit Linux/Unix - Derivaten als unternehmensweite Plattform bedient werden können.	Design-Entscheidung
P3: Benutzerschnittstelle *P3.1 Integrierbarkeit* Die Benutzerschnittstelle soll einfach in beliebige SWT-basierte Oberflächen integriert werden können.	Es ist davon auszugehen, dass die Benutzerschnittstelle im Rahmen der Integration in verschiedenen Bereichen der Anwendungsoberfläche eingesetzt werden wird.	
P4: Zuverlässigkeit Das System muss in hohem Maße ausfallsicher sein.	Systemausfälle sind sehr kostenintensiv und	Design-Entscheidung, umfangreiche Tests

Tabelle 5.1: Produktspezifische Einflussfaktoren

andererseits soll sie sich die SWT-basierte Standard-Variante leicht in existierende Oberflächen, insbesondere des EIF integrieren lassen.

Beeinflussende Faktoren:

P Benutzerschnittstelle integrierbar

O Systemwissen/Erfahrung

T Softwaretechnologien

Lösung: Nutzung von Standards zur Strukturierung von Benutzerschnittstellen sichert die Integrierbarkeit der Benutzerschnittstelle in der Gesamtapplikation und verringert gleichzeitig den Aufwand zur Einarbeitung in neue Technologien und API's. Interfaces auf hohem semantischen Niveau erleichtern den Umgang mit komplexen Datenstrukturen.

Strategie: (Implementierung von Standard-Komponenten) Der Aufbau der Benutzerschnittstelle als autonome SWT-Komponente mit wohldefinierten Schnittstellen ermöglicht einerseits den Einbau in beliebigen SWT-basierten Oberflächen, andererseits setzt die Verwendung nur wenig mehr Wissen als das über den Umgang mit SWT-Standard-Komponenten voraus.

Strategie: (Implementierung als JFace-Viewer) Die Implementierung eines JFace-Viewers als Fassade über einer Oberflächenkomponente erlaubt die Nutzung von Standardtechnologien bei der Anbindung von Datenmodellen.

5.2.3 Systementwurf

Der erste Schritt beim Systementwurf besteht in der Darstellung des Systems mit seinen technischen Komponenten und seinen Schnittstellen zur Umwelt (Abbildung 5.3). Die farblich hervorgehobenen Komponenten werden als Bestandteil des Konfigurations-Frameworks neu zum EIF hinzugefügt. Die WorkflowElementRegistry bildet eine Ausnahme. Sie ist zwar nicht als Komponente hinzugekommen, wurde aber dennoch, wie im vorangegangenen Kapitel beschrieben, vollständig neu konzipiert. In diesem neuen Zustand ist sie essentiell für die Realisierung jeglicher

Funktionalität des neuen Frameworks. Bei der Darstellung wurde in geringem Maße vom UML-Standard abgewichen, um mit Hilfe der use-Beziehung den service-orientierten Charakter zu unterstreichen. Im Rahmen der vorliegenden Arbeit soll

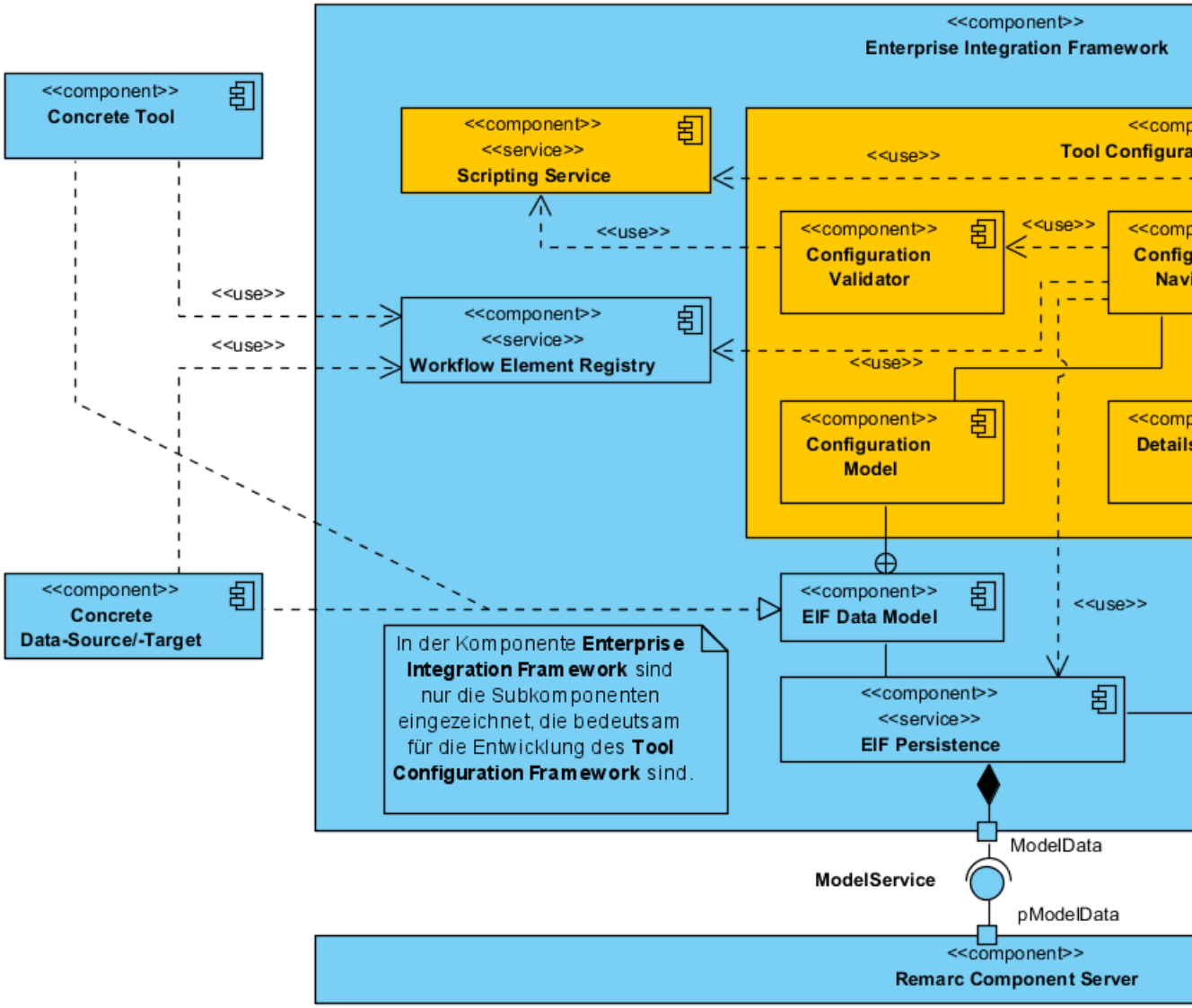

Abbildung 5.3: Systementwurf Configuration Framework

nur auf die Realisierung der in den technologischen Faktoren spezifizierten service-orientierten Architektur eingegangen werden. Als technische Bausteine wurden die identifizierten Bausteine des Analysemodells mit einer Ausnahme übernommen. Der *Datasource-Adapter* wird in die Komponente *Configuration Navigator* integriert. Al-

le Komponenten, mit Ausnahme des Configuration Model und der Oberflächenkomponenten werden als Services ausgebildet. Die Komponenten *Configuration Validator* und *Configuration Value* Interpreter bilden jeweils eine Service-Komposition mit dem *Scripting Service*.

5.3 Umsetzung der erstellten Architektur

Die Umsetzung der erstellten Architektur erfolgt in vier Schritten:

1. Erstellung des Configuration Model
2. Erstellung der serviceorientierten Landschaft
3. Erstellung der Benutzeroberfläche
4. Erstellung der Schnittstellenimplementierung für die zu integrierenden Werkzeuge

Da viele der verwendeten Strategien und Technologien bereits in früheren Kapitel ausführlich erläutert wurden, hat die nachfolgende Beschreibung der einzelnen Implementierungsschritte eher Übersichtscharakter.

Erstellung des Configuration Model Das Datenmodell für die Konfigurationsdaten erweitert das in Kapitel 4.2.1 beschriebene EIF-Datenmodell um ein zusätzliches Package. Damit sind Konfigurationsdaten direkt aus dem Datenmodell des EIF heraus referenzierbar. Bei der Abbildung des geforderten Konfigurationsbaum kommt das Composite-Pattern zum Einsatz.

Definition 5.3.1 (Composite Pattern)**.** *„Füge Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren. Das Kompositionsmuster ermöglicht es Klienten, sowohl einzelne Objekte als auch Kompositionen von Objekten einheitlich zu behandeln.“[Gam+96, S.239]*

Abbildung 5.4 zeigt den Einsatz zur Realisierung des Konfigurationsbaumes. Zu beachten ist, dass die Grafik aus Übersichtsgründen nur ausgewählte Operationen und Attribute beinhaltet, soweit diese jeweils erforderlich sind für die nachfolgenden

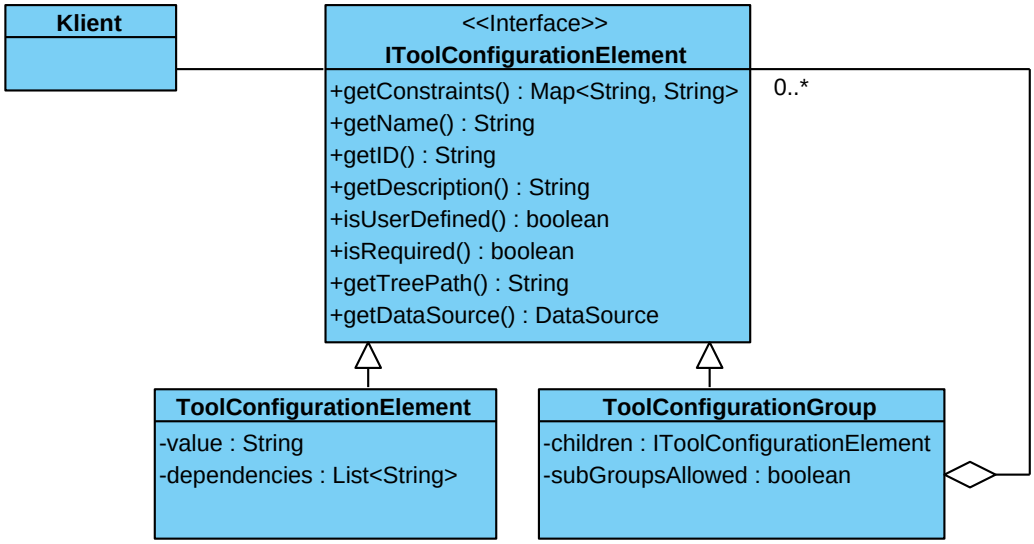

Abbildung 5.4: Klassendiagramm Konfigurationsbaum (qualitativ)

Ausführungen und um Zusammenhänge mit der Anforderungsanalyse herstellen zu können.

Jedes Element des Konfigurationsbaumes (*IToolConfigurationElement*) besitzt neben einem eindeutigen Identifier einen Namen und eine Beschreibung. Diese scheinbare Redundanz ergibt sich aus dem Umstand, dass dem Benutzer die Möglichkeit gegeben werden soll Konfigurationsgruppen-und schlüssel (auch durch Fremdsysteme definierte) mit einem beliebigen Namen zu versehen, ohne die Integrität des Datenmodells zu gefährden. Ein weiterer wichtiger Punkt in diesem Zusammenhang betrifft den Gültigkeitsbereich der ID. Jeder Identifier ist nur innerhalb seines Elternelementes eindeutig bestimmt, d.h. die absolute Eindeutigkeit ergibt sich nach wie vor aus dem kompletten Pfad im aktuellen Konfigurationsbaum (*getTreePath()*). Dies ist vor allem darin begründet, dass Fremdsysteme Schlüssel definieren müssen, ohne andere Integrationsteilnehmer zu kennen.
Die Operation *getDataSource()* gibt je nachdem ob es sich um eine Gruppe oder ein Konfigurationselement handelt, eine Datenquelle zur Ermittlung erlaubter Schlüssel-IDs oder erlaubter Konfigurationswerte zurück.
Das Attribut *dependencies* der Klasse *ToolConfigurationElement* enthält abhängige Schlüssel in Pfad-Schreibweise. Sind derartige Abhängigkeiten definiert, müssen sie vor dem Editieren des *values* durch den *Dependency Solver* gelöst werden.

Die Operation *getConstraints()* einer *ToolConfigurationGroup* liefert alle im Pfad durch den Baum bis zu dieser Stelle definierten Zwangsbedingungen und weist jeder einen Vorgabewert zu, sofern ein solcher ermittelt werden kann. Innerhalb eines *ToolConfigurationElement* liefert diese Funktion die tatsächlich durch den Anwender gesetzten Werte.
Das Datenmodell wurde vollständig modellgetrieben unter Verwendung der entsprechenden Werkzeuge des EMF entwickelt. Das garantiert einerseits die Fähigkeit sich nahtlos in das EIF-Datenmodell einzufügen und damit bereits erstellte Persistenz-Mechanismen nutzen zu können, andererseits die Möglichkeit evolutionär bedingte nachträgliche Änderungen problemlos einfließen zu lassen.
Zur Vereinfachung der Traversierung eines Konfigurationsbaumes oder -waldes wurde eine Utility-Klasse entwickelt, welche die häufigsten Szenarien in diesem Kontext abdeckt. Listing B.2 veranschaulicht den Inhalt dieser Klasse.

Erstellung der serviceorientierten Landschaft Ausnahmslos alle im Entwurf definierten Services werden als deklarative Services im OSGi-Kontext mit der Option der nachträglichen Überführung in *Remote Services*, bei Vorliegen einer entsprechenden Erfordernis, erstellt. Das grundsätzliche Vorgehen wurde allgemein in Kapitel 3.2.3 und am Beispiel der *Workflow-Element-Registry* in Kapitel 4.2.3 erläutert.

Erstellung der Benutzeroberfläche Die Benutzeroberfläche stellt den sensibelsten Bereich des zu entwickelnden Frameworks dar, schließlich ist einer der Hauptgründe für die Beauftragung die komfortablere, übersichtlichere Bereitstellung von Konfigurationsdaten bei gleichzeitiger maximaler Unterstützung des Benutzers bei der Eingabe der Parameter. Die bevorzugte Darstellung für strukturierte Daten, wie sie im Konfigurationsbaum vorkommen, beschreibt das folgende Oberflächenmuster.

Definition 5.3.2 (Two-Panel Selector)**.** *„Put two side-by-side panels on the interface. In the first one, show a list of items that the user can select at will; in the second one, show the content of the selected item.“[Tid11, S. 198]*

Im Eclipse-Umfeld wird dieses Muster auch als *Master-Details-Block* bezeichnet. Der Master-Bereich (links) soll genutzt werden, um den Konfigurations-Navigator

darzustellen, welcher den Konfigurationsbaum enthält, während der Details-Bereich (rechts) das Lesen und Editieren aller Informationen zum jeweils selektierten Element erlaubt. Um die einzelnen Konfigurationsbäume, optisch voneinander zu trennen, wird die Wurzel jeweils durch ein Shelf realisiert. Ein Shelf stellt dabei eine Oberflächenkomponente dar, die einem Schubkasten gleich ihren Inhalt beim Berühren (Öffnen) präsentiert. Es folgt damit dem Muster *List Inlay*.

Definition 5.3.3 (List Inlay)**.** *„Show a list of items as rows in a column. When the user selects an item, open that item's details in place, within the list itself. Allow items to be opened and closed independently of each other.“[Tid11, S.206]*

Ein Kontext-Menü auf jedem Konfigurationsbaum bietet in Abhängigkeit der ausgewählten Elemente und ihrer Eigenschaften erlaubte Operationen an. Abbildung 5.5 zeigt die fertige Implementierung des Navigators.

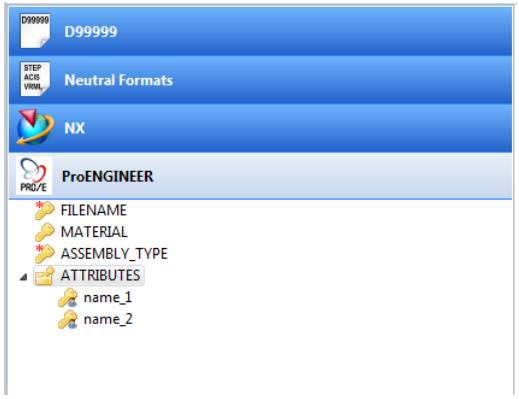

Abbildung 5.5: Konfigurations-Navigator

Als weitere Besonderheit werden Pflichtelemente mit einem roten Stern und nutzerdefinierte Elemente mit einem Nutzersymbol versehen. Dies ermöglicht auf den ersten Blick die erlaubten Operationen einzuschätzen, da alle Beschränkungen sich aus einer Kombination dieser Attribute herleiten lassen. Im Details-Bereich soll ein weiteres Oberflächenmuster zur Anwendung kommen.

Definition 5.3.4 (Titled Sections)**.** *„Define separate sections of content by giving each one a visually strong title, separating the sections visually, and arranging them on the page.“[Tid11, S.153]*

Gemäß diesem Muster wird die Oberfläche des Details-Bereiches in vier Sektionen untergliedert.

1. Bereich für die Auswahl und Konfiguration einer Datenquelle.
2. Bereich für das Editieren/Auswählen des konkreten Wertes.
3. Bereich für das Editieren von Zwangsbedingungen.
4. Bereich für das Editieren von Abhängigkeiten.

Abbildung 5.6 zeigt die Realisierung des Musters. Aus Konformitätsgründen wurde der gesamte Details-Editor ebenfalls in ein Shelf eingebettet, welches allerdings ohne Funktion implementiert wurde. Im Kopf wird weiterhin der Schlüsselname in Punkt-

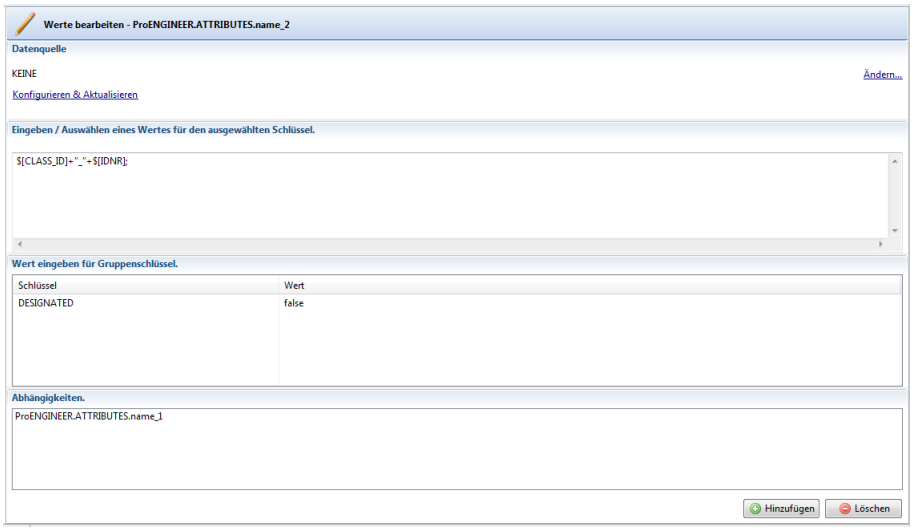

Abbildung 5.6: Details-Editor

schreibweise angezeigt, was gleichzeitig dem vollständigen Pfadnamen entspricht. In der Sektion Abhängigkeiten wird genau über diese Pfadangabe der Wert eines anderen Konfigurations-Schlüssel adressiert. Natürlich muss dieser Pfad nicht manuell eingetragen werden, er wird vielmehr in einem Dialog, mit einer weitgehend identischen Struktur wie der Master-Bereich, zur Auswahl angeboten.
Die Sektion zur Eingabe des Konfigurationswertes wird dynamisch, je nach Auswahl der Datenquelle gestaltet. Verantwortlich hierfür zeichnet die entsprechende *WorkflowElementFactory*, die das aktuell selektierte Element erzeugt hat. Wird keine Datenquelle ausgewählt, erscheint in dieser Sektion ein Texteingabefeld, dessen

Eingabe durch den *ConfigurationValueInterpreter* interpretiert wird.
In der Sektion für die Zwangsbedingungen (Gruppenschlüssel) werden automatisch die geerbten Schlüssel des Elternelementes mit ihren *default*-Werten angezeigt und können durch den Benutzer editiert werden.
Abhängigkeiten werden durch den *Dependency Solver* gelöst. Besitzt ein Schlüssel Abhängigkeiten zu anderen Schlüsseln, versucht diese Komponente rekursiv die Werte der abhängigen Schlüssel zu ermitteln. Gelingt dies nicht, wird der entsprechende abhängige Schlüssel selektiert und der Benutzer zur Eingabe aufgefordert.

Bei der Implementierung wurde großer Wert auf Wiederverwendbarkeit gelegt, wodurch ein hoher Uniformitätsgrad innerhalb des Frameworks erreicht werden konnte. Um der Anforderung der Integrierbarkeit und Austauschbarkeit der Benutzerschnittstelle gerecht zu werden, soll sich die gesamte Oberfläche für den Programmierer als Einheit darstellen. Aus diesem Grund wurden Navigator und Details-Editor als selbständige SWT-Komponenten implementiert. Das bietet den Vorteil, dass Verhalten und Handhabung identisch mit anderen SWT-Komponenten, wie beispielsweise *Button* oder *TextField* ist. Dadurch wird die Einbettung in beliebige Oberflächen problemlos möglich, da alle Standard-Konzepte der SWT-Bibliothek, wie Layout-Manager, greifen. So konnte beispielsweise der Konfigurationsbaum in unterschiedlichen Ausprägungen für die Auswahl der Abhängigkeiten, im Template-Manager (vergleiche Abbildung A.4) für das Laden und Speichern, sowie im Remarc® *New-Wizard* wiederverwendet werden. Im letzteren Fall wird das Anhängen einer als Template gespeicherten Konfiguration bereits zum Zeitpunkt der Erstellung einer neuen Normreihe ermöglicht.
Ein weiterer Vereinfachung, vor allem im Umgang mit dem Datenmodell, liegt in der Bereitstellung einer High-Level-API. Dieses Konzept setzt die Standard-Bibliothek *jFace* für SWT-Komponenten um. Die Implementierung eines sogenannten *Viewers*, welcher die Konfigurationskomponente beinhaltet, setzt die Bereitstellung eines *ContentProviders* und eines *LabelProviders* voraus. Während der *ContentProvider* das an einen *Viewer* übergebene Datenobjekt zur Darstellung strukturiert aufbereitet, bestimmt der *LabelProvider* die Repräsentationsform an der Oberfläche. Die Implementierung dieses Konzeptes muss aus Komplexitätsgründen einer späteren Version des Frameworks vorbehalten bleiben.

Schnittstellenimplementierung Für die Realisierung des Prototyps ist die Implementierung entsprechender *DataSources*, zum einen zur Erstellung des Konfigurationsbaumes für jedes Werkzeug, zum anderen zur Bereitstellung der Wertelisten aus Teamcenter erforderlich. Hierfür erweitert jede der zu integrierenden Anwendungen das Datenmodell um eine konkrete *DataSource* und implementiert die dazugehörige Methode *getData()*. Damit die Konfiguration abgefragt werden kann, muss das EIF-Datenmodell um die Operationen *getConfiguration()* und *setConfigurationToDefault()* für *Tools* ergänzt werden. Diese Methoden werden aufgerufen um den jeweiligen Konfigurationsbaum abzufragen und mit Vorgabewerten zu befüllen. Im Falle der DataSource für Wertelisten muss natürlich noch die Deklaration der Erweiterung des entsprechenden Extension Point durchgeführt werden.

Organisatorische Faktoren	Flexibilität und Veränderbarkeit	Einfluss
O1: Mitarbeiter		
O1.1 Wissen im Bereich CAD		
Wissen im Bereich spezifischer CAD-Systeme und zugehöriger APIs ist weitgehend vorhanden.	Wissensaufbau für andere CAD-Systeme ist bei entsprechendem Bedarf durch konkrete Anforderungen (neue Projekte) möglich.	Zeitplan, Design-Entscheidung
O1.2 Systemwissen/Erfahrung		
Hohe Mitarbeiterfluktuation verursacht Wissensverluste.	Der Einsatz von häufig wechselndem Entwicklungspersonal (Studenten) im Entwicklungsteam führt zu Verlust von Wissen über das System. Mangelnde Erfahrung erschwert die Einhaltung einer sauberen Architektur.	Zeitplan, Design-Entscheidung
O2: Entwicklungszeitplan		
O2.1 Release-Plan		
Der Termin für das nächste Release steht fest.	Terminverschiebungen sind in begründeten Fällen möglich.	Zeitplan und Design-Entscheidung
O3: Entwicklungsbudget		
O3.1 Entwicklungskosten		
Entwicklungskosten müssen überschaubar bleiben.	Auf externe Kapazitäten darf nur im Ausnahmefall (Grafiker) zurückgegriffen werden. Ein kleines Budget für Weiterbildungsmaßnahmen ist eingeplant.	Zeitplan und Designentscheidung

Tabelle 5.2: Organistorische Einflussfaktoren

Technologische Faktoren	Flexibilität und Veränderbarkeit	Einfluss
T1: Hardware		
T1.1 Prozessor		
Intel/AMD 32/64-Bit	Ist von der zu verwendenden Java Runtime 1.5 abhängig	geringer Einfluss, da die Voraussetzungen durch Multi-Cad-Umgebungen, welche das vordergründige Ziel bilden, übertroffen werden.
T1.2 Speicher		
64/128 MB	Ist von der zu verwendenden Java Runtime 1.5 und dem Betriebssystem abhängig	geringer Einfluss, da die Voraussetzungen durch Multi-Cad-Umgebungen übertroffen werden.
T2: Softwaretechnologien		
T2.1 Betriebssystem		
Als Betriebssystem wird Windows ab Version XP vorausgesetzt.	In zukünftigen Versionen kann sich das Betriebssystem ändern. Für das nächste Release ist es aber festgelegt.	Design-Entscheidung
T2.2 Programmiersprache		
Als Programmiersprache ist Java ab V1.5.x festgelegt.	Eine Änderung der Programmiersprache ist nicht möglich, da das Framework teil des EIF wird und dieses bereits zu großen Teilen in dieser Sprache implementiert wurde.	Design-Entscheidung
T2.3 Frameworks		
Als Basis wird die Eclipse-RCP V3.7 und die zugehörigen EMF-Features festgelegt.	Die Frameworks sind nicht verhandelbar, da die Integration ins EIF sichergestellt werden muss.	Design-Entscheidung und Zeitplan

Tabelle 5.3: Technologische Einflussfaktoren I

Technologische Faktoren	Flexibilität und Veränderbarkeit	Einfluss
T3: Architekturtechnologien		
T3.1 Architektustile/-muster		
Objektorientiertheit,Implicit Invocation, Schichten, Eventgesteuertes System, Repository, Serviceorientierte Architektur	Mindestmaß an zu verwendenden Architekturstilen/-mustern, kann durch spätere Anforderungen ergänzt werden	Design-Entscheidung
T3.2 Produktlinienansatz		
Domain-Framework, stark modularisiert	Über verwendete Module wird das gelieferte Produkt definiert.	festgelegt, kein Entscheidungsspielraum
T4: Standards		
T4.1 Datenbanken		
Oracle,MySQL,...	Datenbanksysteme sind kundenspezifisch und müssen problemlos integrierbar sein.	festgelegt, Design-Entscheidung

Tabelle 5.4: Technologische Einflussfaktoren II

6 Zusammenfassung und Ausblick

Erfolg besteht darin, dass man genau die Fähigkeiten hat, die im Moment gefragt sind.

(Henry Ford, 1863-1947)

In diesem Kapitel soll das Resultat der Arbeit in komprimierter Form dargestellt und der in der Zielsetzung definierten Aufgabenstellung gegenübergestellt werden. Insbesondere werden Herausforderungen und abgeleitete Erkenntnisse reflektierend diskutiert und die weiteren Entwicklungsschritte skizziert.

6.1 Darstellung der wichtigsten Ergebnisse

Erklärtes Ziel der Arbeit war die Entwicklung eines Frameworks, welches als eigenständige Schicht im EIF Daten von unterschiedlichen Quellen beziehen und sie integrierten Anwendungen zur Parametrisierung bzw. Konfiguration ihrer Schnittstellen zur Verfügung stellen kann. Insbesondere sollten auch die dynamische Ermittlung Laufzeit-orientierter Parameter und die Auflösung von Abhängigkeiten zwischen Konfigurationswerten möglich sein.
Im Rahmen der vorliegenden Arbeit konnte eine Architektur für dieses Framework erstellt werden, deren Tragfähigkeit durch prototypische Implementierung des Systems und Realisierung eines konkreten Integrations-Workflows nachgewiesen wurde, der die Systeme Remarc® CSM, Siemens NX und Siemens Teamcenter miteinander verbindet.
Dieser Nachweis konnte nach Abschluss der Implementierung erfolgreich erbracht werden.

Zentraler Punkt des Systems ist die Benutzerschnittstelle, der Konfigurationsnavigator. Dieser stellt Konfigurationsbäume, die von jedem integrierten Softwarewerkzeug zur Verfügung gestellt werden müssen dar. Anwender sind dann in der Lage weitere Konfigurationsschlüssel oder ganze Gruppen hinzuzufügen und allen Konfigurationsschlüsseln manuell Werte zuzuordnen oder, durch Auswahl bzw. Definition geeigneter Datenquellen, passende Werte eines bestimmten Werteraumes zuzuweisen. Insbesondere manuell eingegebene Werte können Skript-Charakter haben und durch ihre Interpretierbarkeit dynamische Aspekte der Parametrisierungsanforderungen umsetzen. Die Interpretation übernimmt dabei eine Service-Komposition, die den aktuell benötigten Interpreter-Typ ermittelt und den Wert interpretiert. Hierfür wurden teilweise vorhandene Programmstrukturen erweitert und in Services überführt. Die Anbindung von Datenquellen zur Ermittlung von Werteräumen konnte erfolgreich umgesetzt werden. Die Nutzung dieses Konzeptes zur Bereitstellung von Schlüsselräumen ist hingegen, abgesehen von der initialen Bereitstellung des Konfigurationsbaumes, noch nicht realisiert worden.

Die größte Herausforderung bei der Integration von Anwendungen unter Nutzung des EIF liegt, nicht zuletzt aufgrund der Neueinführung des Konfigurations-Frameworks, in der Nutzung der API des Integrationsteilnehmers. So ist es möglich mit nur wenigen Schritten eine neue Applikation zu integrieren.

1. Anlegen eines Plugins, mit entsprechenden Abhängigkeiten zum EIF und zur API des Integrationsteilnehmers
2. Definition der Extension für ein WorkflowElement (Tool, DataSource, DataTarget) und optionaler Kategorien
3. Implementierung der entsprechenden Schnittstellen-Realisierung. Hier kommt es vor allem auf Kenntnisse der fremden API an.
4. Definition einer DataSource zur Bereitstellung des Konfigurationsbaumes.

Aufgrund dieser Einfachheit konnte zumindest das Konfigurations-Framework vollständig und produktiv zur Konfiguration und Parametrisierung aller Prozesse des Remarc® Komponenten-Frameworks eingesetzt werden. Es hat aber auch gezeigt, dass das Konzept grundsätzlich zur Integration beliebiger Anwendungen, unabhängig von der Domäne, geeignet ist. Es beschränkt sich jedoch im Wesentlichen auf den

Einsatz innerhalb einer Eclipse-RCP. Im Zusammenspiel mit dem vollständigen EIF entfällt diese Beschränkung unter der Voraussetzung, dass die Schnittstellenimplementierung strukturell ein Eclipse-Plugin darstellt. Soll das Konfigurationsmodell am Domänenmodell gespeichert werden, muss dieses mit EMF erstellt worden sein oder die Zuordnung der Business-Entität zum Konfigurationsbaum muss durch programmatischen Eingriff in dieses Modell erfolgen.
Mit der Arbeit konnte weiterhin gezeigt werden, dass sich OSGi durchaus zum Aufbau serviceorientierter Architekturen eignet und es damit möglich ist, insbesondere in Verbindung mit EMF und CDO, konkurrenzfähige, hochmodulare Unternehmensanwendungen zu entwickeln.

6.2 Weitere Entwicklungsschritte

Das Konfigurations-Framework hat trotz der relativ kurzen Entwicklungszeit bereits in einen relativ hohen Reifegrad erreicht. Die implementierten Funktionen verhalten sich stabil und bereits durchgeführte sehr umfangreiche Tests förderten keine nennenswerte Fehler zutage. Neue Werkzeuge lassen sich innerhalb kurzer Zeit einbinden. Diese Eigenschaften beweisen die Tragfähigkeit und die konzeptionelle Stärke der erarbeiteten Architektur.
Die nächsten Meilensteine in der Entwicklung sollten die nachfolgenden Punkte abdecken, um den Bedienkomfort weiter zu verbessern und Fehlbedienungen noch besser auszuschließen.

- Nutzung von Datenquellen zur Definition von Schlüsselräumen.
- Möglichkeit der nachträglichen Änderung von Schlüssel- und Gruppennamen.
- Integration der Konfiguration in den EIF-Properties-View.
- Implementierung des Konfigurations-Validators
- Erweiterung des Eingabefeldes für Werte um Möglichkeiten der Editierunterstützung, beispielsweise Syntax-Highlighting oder Code-Vervollständigung.

Diese Auswahl soll ausreichend sein, das Entwicklungspotential aufzuzeigen.

Glossar

CRUD-Operationen bezeichnet die grundlegenden Datenbankoperationen, *CREATE*, *READ*, *UPDATE*, *DELETE*.. 14

Extension Point Extension Points geben die Möglichkeit ein Bundle für Erweiterungen zu öffnen. Es ist ein historisch gewachsenes Konzept in einer Eclipse-Anwendung und basiert nicht auf OSGi. Der Extension Point beschreibt deklarativ die Anforderungen an die Erweiterung. Die konkreten Erweiterungen müssen somit zur Entwicklungszeit nicht bekannt sein.. 48, 57

Plugin Ein Plugin stellt im Sinne dieser Arbeit ein OSGi-Bundle der Equinox-Distribution dar.. 48

Service Level Agreement bezeichnet eine Vereinbarung zwischen Service Provider und Service Consumern zur Spezifizierung des diesen minimal zur Verfügung stehenden Services.[Hil00]. 26

Service-Inventar Ein *Service Inventar* bezeichnet eine Sammlung komplementärer Services in einem Unternehmen oder dedizierten Unternehmensbereich. Es ist ein Indikator für den Grad der Umsetzung einer SOA in einem Unternehmen.[Erl08]. 30

SOAP bezeichnet ein Netzwerkprotokoll zum *XML*-basierten Datenaustausch zwischen verschiedenen Systemen. Es stützt sich auf Transport- und Anwendungsebene auf die verbreiteten Protokolle des *TCP/IP-Referenzmodells*. Insbesondere kommen *HTTP* auf Anwendungsebene und *TCP* auf Transportebene zum Einsatz.. 41

Universal Description, Discovery and Integration (UDDI) Defining a standard method for enterprises to dynamically discover and invoke Web services.[OAS]. 32

WSDL bezeichnet eine XML-basierte Beschreibungssprache für Webservices. Sie ist in hohem Maße plattform-, protokoll- und sprachunabhängig.. 41

XA Transaction Standard für die Verarbeitung von verteilten Transaktionen, welcher durch die *Open Group*[27] spezifiziert wurde und auf dem *Zwei-Phasen-Commit-Protokoll* beruht.. 22

Zwei-Phasen-Commit-Protokoll bezeichnet ein Protokoll zur Durchführung von Transaktionen in verteilten Systemen. In der ersten Phase, die auch als Abstimmungsphase bezeichnet wird, holt ein koordinierender Prozess die Zustimmung aller Teilprozesse. Wird diese ausnahmslos erteilt, erfolgt in der zweiten Phase, der Commit-Phase, ein *Commit*. Verweigert auch nur ein Prozess die Zustimmung wird für alle ein *Rollback* angeordnet.[KE06]. 22

[27] Zusammenschluss der *Open Software Foundation* und der *X/Open*, `http://www3.opengroup.org/`

Anhang A

Abbildungen/Screenshots

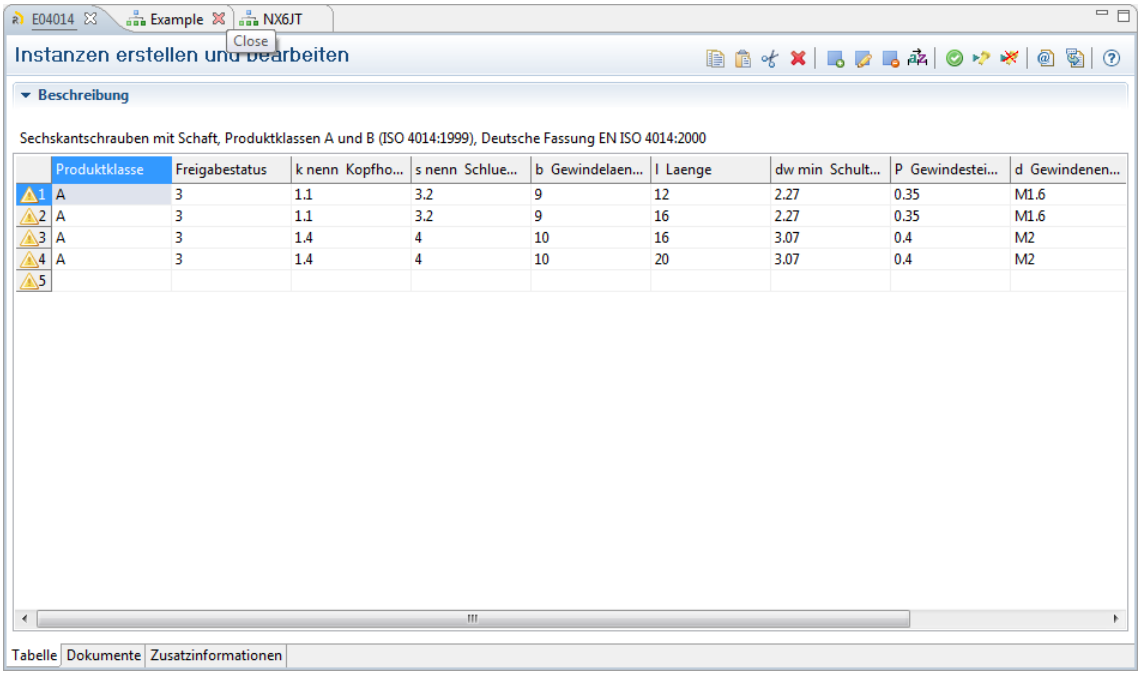

Abbildung A.1: Mapping-Editor

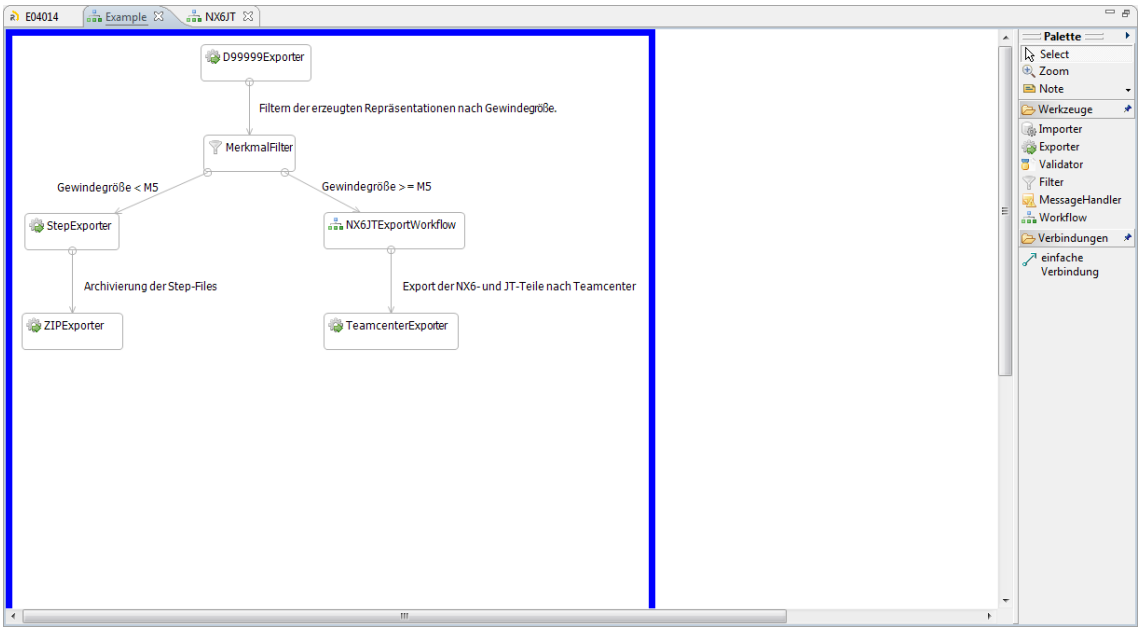

Abbildung A.2: Workflow-Editor

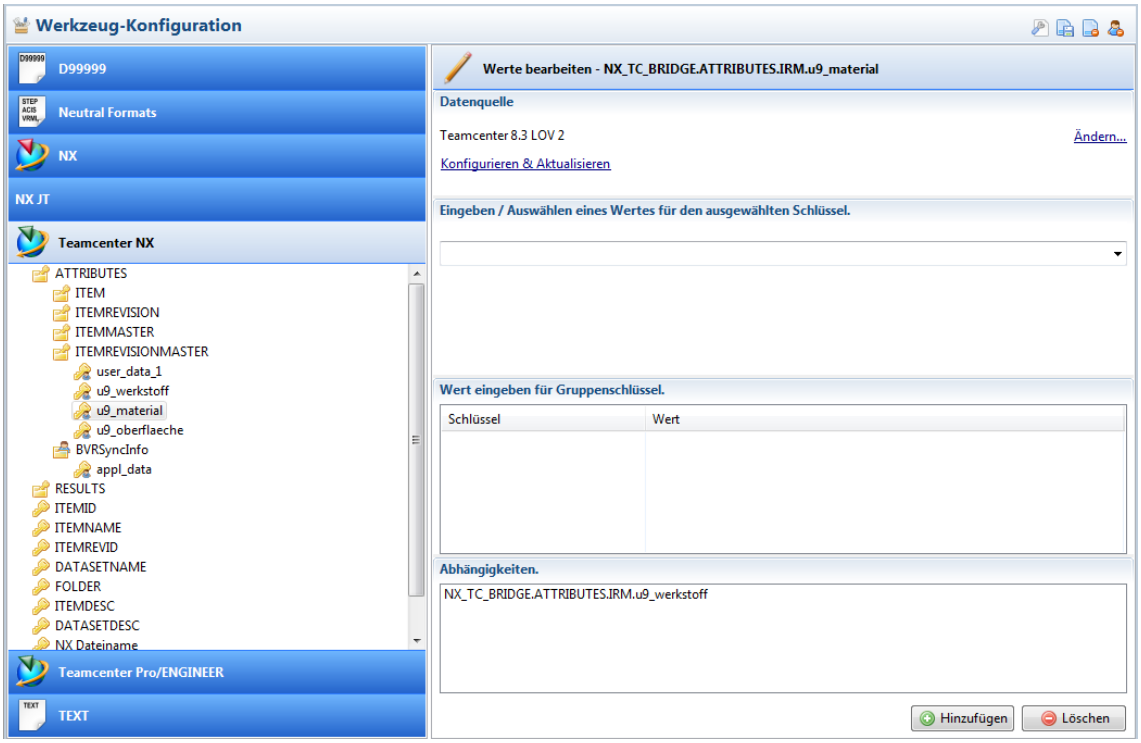

Abbildung A.3: Oberflächenkomponente des Konfigurations-Framework

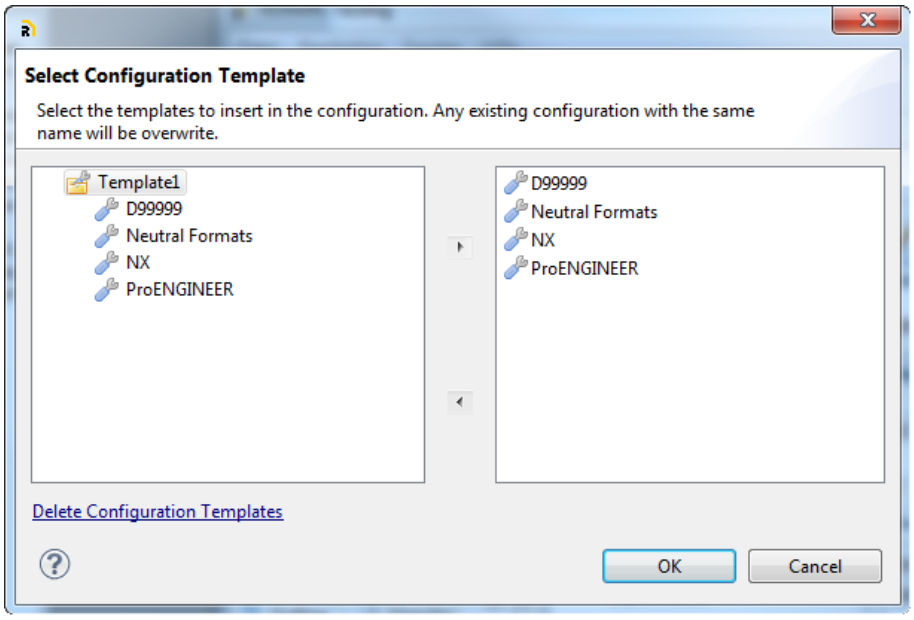

Abbildung A.4: Dialog zum Laden von Konfigurations-Templates

Anhang B

Details der Implementierung

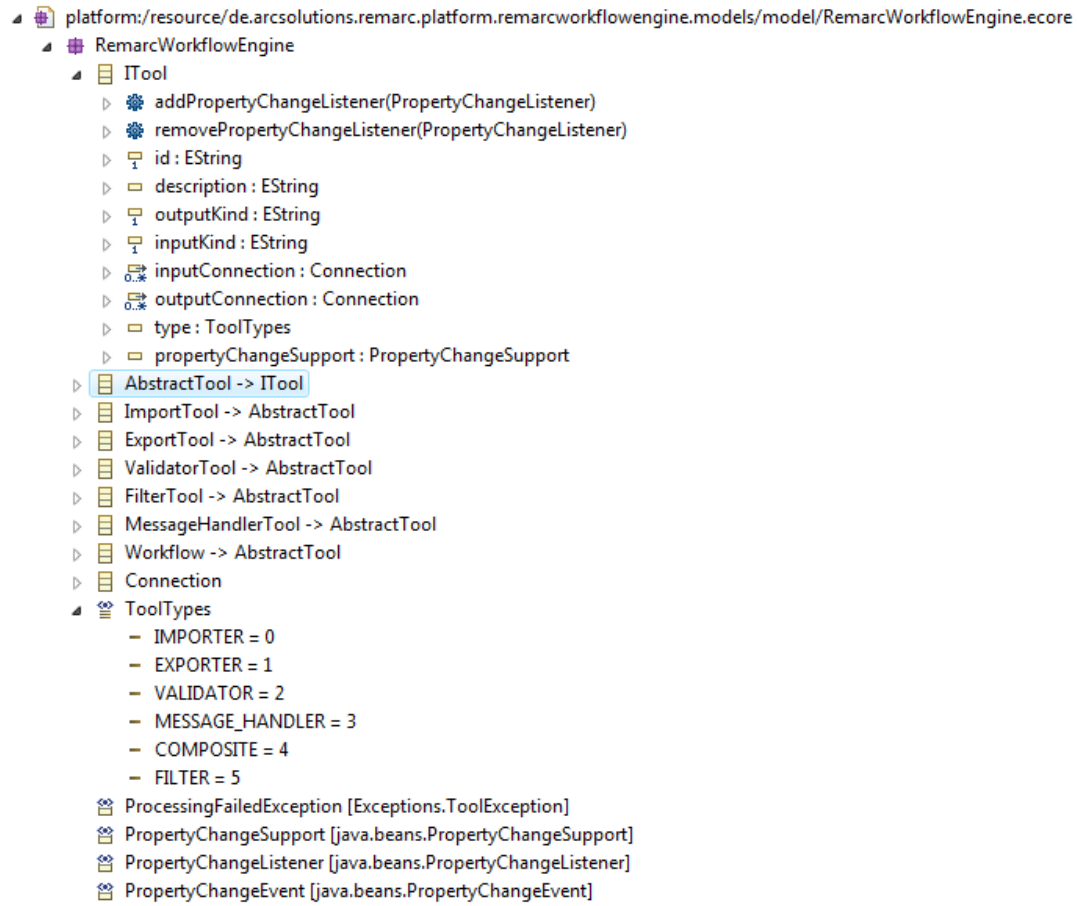

Abbildung B.1: Abstrakte Syntax Definition eines Workflows (ecore)

```
public interface WorkflowElementRegistry {

  /**
   * The {@link SearchType} determines a special attribute that will be
       used
   * as a criterion for searching {@link IConverter}.
   *
   * @author ms
   */
  public enum SearchType {
    ID, INPUT_KIND, OUTPUT_KIND, CATEGORY, CLASS, ELEMENT_TYPE, ALL
  }

  public enum ElementType {
    DATA_SOURCE, DATA_TARGET, TOOL, WORKFLOW
  }

  /**
   * Gets the ids of all available {@link IWorkflowElement}s provided
       by both
   * extension point and file system.
   *
   * @return - the available ids
   * @throws CommitException
   */
  Long[] getAvailableWorkflowElementIDs()
      throws CommitException;

  /**
   * Gets all available {@link IWorkflowElement} filtered by given
       parameter
   * set provide by both extension point and file system.
   *
   * @param type
   *            - specifies the type of the constraint
   * @param value
   *            - the value of the constraint
   * @return - all {@link IWorkflowElement} that have an attribute
       appropriate
   *         to the given set - null if no element has matched
```

```
     * @throws CommitException
     */
    List<WorkflowElement> findWorkflowElement(SearchType type, String
        value);

    /**
     * Gets all available Categories, provided by Extension Point
     * de.arcsolutions.remarc.eif.workflowelement.category
     *
     * @return all Categories found
     */
    List<String> getCategories();

    /**
     * Gets each {@link WorkflowElementFactory} that was already used.
     *
     * @param type
     *            the ElementType which the Factories should be able to
        create,
     *            or null for all Factories
     * @return a Collection of {@link WorkflowElementFactory}
     */
    List<WorkflowElementFactory<?>> getWorkflowElementFactories(String
        type);
}
```

Listing B.1: Das Service-Interface WorkflowElementRegistry

```
public class ToolConfigurationUtil
{

  public static ToolConfigurationElement
    getConfigurationElementById(
    final ToolConfigurationGroup root,
    final String id){}

  public static ToolConfigurationGroup
    getConfigurationGroupById(
    final ToolConfigurationGroup root, final String id){}

  public static ToolConfigurationGroup
    getConfigurationGroupById(
    final List<ToolConfigurationGroup> roots,
    final String id){}

  public static ToolConfigurationElement
      getToolConfigurationElementFromRootById(
    final List<ToolConfigurationGroup> roots,
    final String rootId,
    final String elementId){}

  public static ToolConfigurationGroup
      getToolConfigurationGroupFromRootById(
    final List<ToolConfigurationGroup> roots,
    final String rootId,
    final String groupId){}

  public static List<ToolConfigurationElement>
      getElementsFromToolConfigurationGroup(
    final ToolConfigurationGroup group){}

  // Tiefensuche
  public static List<ToolConfigurationElement>
      getAllElementsFromToolConfigurationGroup(
    ToolConfigurationGroup group){}

  public static List<ToolConfigurationGroup>
      getSubGroupsFromToolConfigurationGroup(
    final ToolConfigurationGroup group){}

  public static ToolConfigurationElement
    getConfigurationElementByPath(
    final List<ToolConfigurationGroup> groups,
    final String configTreePath){}
}
```

Listing B.2: Utility-Klasse für Konfigurationsbäume (gekürzt)

Literaturverzeichnis

Computergestützte Konstruktion

[Bug96] Rainer Bugow. *Die Bereitstellung von Teilebibliotheken im rechnerunterstützten Konstruktionsprozeß*. 1. Auflage. DIN Deutsches Institut für Normung: DIN-Normungskunde. Beuth Verlag GmbH, 1996.

[Spr09] Michael Spranger. „Entwurf und prototypische Implementierung eines Domain-Frameworks zur Integration und Steuerung von Softwarekomponenten am Beispiel des Modelldatenaustauschs zwischen diversen CAD-Systemen“. Bachelorarbeit. Mittweida: University of Applied Sciences Mittweida, 2009 (siehe S. 1, 45).

Product Lifecycle Management

[CIM11a] Inc. CIMdata. *CIMdata 2011 Market Analysis Report Series - 2011 PLM Industry Review and Trends Report*. Juli 2011. URL: `http://www.cimdata.com`.

[CIM11b] Inc. CIMdata. *CIMdata 2011 Market Analysis Report Series - 2011 PLM Market and Solution Supplier Analysis Report*. Sep. 2011. URL: `http://www.cimdata.com` (siehe S. 11, 16).

[ES09] Martin Eigner und Ralph Stelzer. *Product Lifecycle Management - Ein Leitfaden für Product Development und Life Cycle Management*. 2. Auflage - Berlin Heidelberg. Springer Verlag, 2009 (siehe S. 9, 10).

[Gri06] Michael Grieves. *Product Lifecycle Management: Driving the Next Generation of Lean Thinking*. Mcgraw-Hill Professional, 2006.

[RNS94] Ulrich Rembold, Bartholomew O. Nnaji und Alfred Storr. *CIM: Computeranwendung in der Produktion*. 1. Auflage - Deutschland. Addison Wesley, 1994 (siehe S. 15).

[VDM08] VDMA. *Leitfaden zur Erstellung eines unternehmensspezifischen PLM-Konzeptes*. 1. Auflage - Frankfurt/M. VDMA Verlag, 2008 (siehe S. 10, 11).

Eclipse RCP

[Bra+09] Christian Brand u. a. „Graphiti - Entwicklung hochwertiger grafischer Editoren“. In: *Eclipse Magazin* 06 (2009), S. 33–36 (siehe S. 56, 58).

[CR09] Eric Clayberg und Dan Rubel. *eclipse Plug-ins*. 2. Auflage - Boston. Addison Wesley, 2009.

[Dau06] Berthold Daum. *Das Eclipse-Codebuch*. 1. Auflage - Heidelberg. dpunkt.verlag, 2006.

[Dau08] Berthold Daum. *Rich-Client-Entwicklung mit Eclipse 3.3*. 3. Auflage - Heidelberg. dpunkt.verlag, 2008.

[Fou] Eclipse Foundation. *Model To Text (M2T)*. URL: `http://www.eclipse.org/modeling/m2t/`.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project - A Domain-Specific Language (DSL) Toolkit*. 1. Auflage - Boston. Addison-Wesley, 2009.

[Ste+09] Dave Steinberg u. a. *EMF Eclipse Modeling Framework*. 3. Auflage - Boston. Addison-Wesley, 2009 (siehe S. 20).

[Tid11] Jenifer Tidwell. *Designing Interfaces*. 2. Auflage - Canada. O'Reilly Verlag, 2011 (siehe S. 75, 76).

Softwaretechnik /-architektur

[Bal00] Helmut Balzert. *Lehrbuch der Softwaretechnik - Software-Entwicklung*. 2. Auflage. Bd. 1. Spektrum Akademischer Verlag, 2000.

[FH08] Eric Freeman und Elisabeth Hartmann Freeman. *Entwurfsmuster von Kopf bis Fuß*. 1. Auflage/4.korrigierter Nachdruck - Köln. O'Reilly Verlag, 2008.

[Gam+96] Erich Gamma u. a. *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software*. 1. Auflage. Addison Wesley, 1996 (siehe S. 38, 53, 54, 73).

[HT03] Andrew Hunt und David Thomas. *Der Pragmatische Programmierer*. 1. Auflage. Hanser Verlag, 2003.

[KE06] Alfons Kemper und André Eickler. *Datenbanksysteme - Eine Einführung*. 2. Auflage. Oldenbourg Wissenschaftsverlag, 2006 (siehe S. 88).

[Kec06] Christoph Kecher. *UML 2.0 - Das umfassende Handbuch*. 2. Auflage Bonn. Gallileo Press, 2006.

[Sch09] Prof. Dr.-Ing. Uwe Schneider, Hrsg. *Scientific Reports - Journal of the University of Applied Sciences Mittweida - Informatik (11.Informatik-Tag)*. Mittweida: Hochschule Mittweida, 2009.

[SH09] Michael Spranger und David Hein. „Modellgetriebene Entwicklung von Softwaremodulen auf Basis von Eclipse". In: Hrsg. von Prof. Dr.-Ing. Uwe Schneider. Mittweida: Hochschule Mittweida, 2009, S. 10 –12.

[SH11] Gernot Starke und Peter Hruschka. *Software-Architektur kompakt - angemessen und zielorientiert*. 2. Auflage - Heidelberg. Spektrum Akademischer Verlag, 2011.

[Ste] Eicke Stepper. *CDO Model Repository Overview*. URL: `http://www.eclipse.org/cdo/documentation/` (siehe S. 21).

[SVE07] Thomas Stahl, Markus Völter und Sven Efftinge. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. 2. Auflage. d.punkt Verlag, 2007 (siehe S. 19).

[ZG10] Peter Zöller-Greer. *Software-Architekturen - Grundlagen und Anwendungen*. 3. Auflage - Wächtersbach. Composia Verlag, 2010 (siehe S. 24).

SOA und OSGi

[Erl08] Thomas Erl. *SOA - Entwurfsprinzipien für serviceorientierte Architektur*. 1. Auflage - München. Addison Wesley, 2008 (siehe S. 23, 25, 26, 28–33, 87).

[Hil00] Andrew Hiles. *Service Level Agreements - Winning a Competitive Edge for Support & Supply Services*. 2. überarbeitete Auflage. Rothstein Associates Inc., 2000 (siehe S. 87).

[Jos08] Nicolai Josuttis. *SOA in der Praxis - System-Design für verteilte Geschäftsprozesse*. 1. Auflage - Heidelberg. dpunkt Verlag, 2008 (siehe S. 25–27, 33).

[MVA10] Jeff McAffer, Paul VanderLei und Simon Archer. *OSGi and Equinox - Creating Highly Modular Java Systems*. 1. Auflage Boston. Addison Wesley, 2010 (siehe S. 34).

[OA a] OSGi-Alliance. *OSGi Service Platform Core Specification, Release 4, version 4.3*. 4.1 - 2011. URL: `http://www.osgi.org/download/r4v43/r4.core.pdf` (siehe S. 34, 37, 38).

[OA b] OSGi-Alliance. *OSGi Service Platform Service Compendium, Release 4, version 4.2*. 8.1 - 2009. URL: `http://www.osgi.org/download/r4v42/r4.cmpn.pdf` (siehe S. 34, 41, 42).

[OA c] OSGi-Alliance. *RFC 119 - DistributedOSGi*. 8.6 - 2008. URL: `http://www.osgi.org/download/osgi-4.2-early-draft.pdf` (siehe S. 44).

[OAS] OASIS. *OASIS UDDI Specification TC*. URL: `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uddi-spec` (siehe S. 88).

[WBB10] Bernd Weber, Patrick Baumgartner und Oliver Braun. *OSGi für Praktiker - Prinzipien, Werkzeuge und praktische Anleitungen auf dem Weg zur „Kleinen SOA“*. 1. Auflage München Wien. Carl Hanser Verlag, 2010.

[Wüt+08] Gerd Wütherich u. a. *Die OSGi Service Platform - Eine Einführung mit Eclipse Equinox*. 1. Auflage - Heidelberg. dpunkt.verlag, 2008 (siehe S. 34).

Schriftliche Versicherung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Mittweida, den 02.01.2012